Intelligent packet error prediction for enhanced radio network performance

Muhammad Irfan Ali

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.
Aalto University 28.7.2019

Supervisor

Prof. Simo Särkkä

Advisor

Dr. Dani Korpi

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Muhammad Irfan Ali

Title Intelligent packet error prediction for enhanced radio network performance

Degree programme Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Prof. Simo Särkkä

Advisor Dr. Dani Korpi

Date 28.7.2019 **Number of pages** 83+1 **Language** English

Abstract

In cellular communication systems, for example 4G and 5G, quite often data packets (in user-plane payload) fail to successfully deliver to the user equipment (UE). Because upon failure, a re-transmission of the data packet is required by the network, these failed data packets introduce latency to the network. In some applications, such latency might be tolerable by the UE, but in applications that require ultra reliable low latency communication (URLCC), time latency becomes a critical issue. In order to cope with this issue, typically wireless networks rely on re-transmissions upon receiver request or use naïve approach like packet duplication to transmit data packets more than once to ensure successful transmission of at least one data packet without any error.

In this thesis, we explore the feasibility of designing an intelligent solution to this issue by using network data with machine learning and neural networks to predict if a data packet would fail to transmit in the next transmission time interval (TTI). Our research includes a detailed systematic study on which radio parameters to choose from the raw data (log files) and data preprocessing. From our experiments we also determine how many past values of these radio parameters can be useful to predict the packet failure in the next TTI. Moreover, we enlist the network parameters useful to make such a prediction and compare their contribution in the model. Finally, we show that an intelligent packet error prediction can be done using machine learning that forecasts the packet failure in the next TTI with sufficient accuracy.

We compare the performance of different machine learning algorithms and show that boosted decision trees (XGBoost) perform the best on the given dataset. Compared to naïve approaches used in cellular communication to avoid packet failures, our solution based on intelligent packet error prediction indicates promising practical applications in cellular network for enhanced radio network performance, particularly in URLLC.

Keywords intelligent packet error prediction, packet error prediction with machine learning, enhanced radio network performance with neural networks

Preface

I would like to thank Prof. Simo Särkkä for the help with thesis writing and reviewing. Secondly, I would like to thank Dr. Dani Korpi for the help and guidance with not only thesis writing and reviewing but also for technical insights into communication and machine learning. I would also like to thank Dr. Mikko Uusitalo, Dr. Zhi Chun and all other Nokia colleagues who helped me during my thesis one way or another. Finally, I would like to dedicate this thesis to my parents and my elder brother Dr. Muhammad Mubashir Nawaz who have always, unconditionally, supported me and believed in me.

Aalto University, 28.7.2019

Muhammad Irfan Ali

Contents

Symbols and Abbreviations

Symbols

b	bias of the underlying model
$\tilde{\boldsymbol{C}}_t$	cell memory of LSTM at time t
δ	error
eta	shrinkage factor in tree boosting
$\boldsymbol{f}_t$	forget gate vector in LSTM at time t
$\mathcal{F}$	set of all possible CARTs
g	first order derivative of the loss function
$g(.)$	smooth and bounded function such as the logistic sigmoid
h	second order derivative of the loss function
$\boldsymbol{h}_t$	recurrent hidden state in RNN at time t
$\tilde{\boldsymbol{h}}_t$	new memory cell content in GRU at time t
$\boldsymbol{i}_t$	input gate vector in LSTM at time t
$\mathcal{L}(.)$	training loss function
O	Output of neural network
$O(.)$	Objective function
$\boldsymbol{o}_t$	output gate vector in LSTM at time t
$\Omega(.)$	regularization function
$\phi(.)$	non-linear activation function
$\boldsymbol{r}_t$	reset gate vector in GRU at time t
$\sigma(.)$	sigmoid function
$\mathbf{w}$	weight vector of the underlying model
$\mathbf{X}$	feature vector
$\mathbf{y}$	ground truth vector
$\hat{\mathbf{y}}$	predicted labels
$\boldsymbol{z}_t$	update gate vector in GRU at time t

Operators

$\mathbf{A} \cdot \mathbf{B}$	dot product of vectors $\mathbf{A}$ and $\mathbf{B}$
$\frac{\mathrm{d}}{\mathrm{d}t}$	derivative with respect to variable t
$\frac{\partial}{\partial t}$	partial derivative with respect to variable t
$\sum_i$	sum over index i
$\odot$	Hadamard product

Abbreviations

2G	Second-Generation Wireless
3G	Third-Generation Wireless
5G	Fifth-Generation Wireless
ACK	Acknowledgement (of data packages)
AI	Artificial Intellgience
ARQ	Automatic Repeat Request
CART	Classification and Regression Trees
CRC	Cyclic Redundancy Check
CSV	Comma Separated values
ELU	Exponential Linear Unit
FEC	Forward Error Correction
eNBs	Evolved Node Bs
GRU	Gated Recurrent Unit
HARQ	Hybrid Automatic Repeat Request
LSTM	Long Short-Term Memory
LTE	Long-Term Evolution (4G mobile communication standard)
MAC	Medium Access Layer
ML	Machine Learning
MLP	Multilayer Perceptron Model
NACK	Non-acknowledgement (of data packages)
NN	Neural Network
RNC	Radio Network Controller
RNN	Recurrent Neural Network
TTI	Transmission Time Interval
UE	User Equipment
URLCC	Ultra Reliable Low Latency Communication

1 Introduction

In a cellular network, communication takes places between a base station and a user equipment (UE) where data packets are transferred between the user and the network. After receiving the data packets, UE sends a flag signal back to the network indicating whether the transmission was successful or not. In LTE (long-term evolution) or 4G for instance, data packets are sent to the user in the form of transport block (payload of UE in user-plane) (Ali-Yahiya, 2011). These transport blocks are padded with cyclic redundancy check (CRC, (Enns and O'Hare, 1991)) which helps the UE with error detection. After receiving the transport block, the UE performs CRC check and sends back the error detection result to the network. If data packet was delivered successfully, the UE sends *acknowledgement* (ACK) to the network indicating that it is ready to receive another transport block. On the contrary, if the receiver detects an error while decoding the data packet, it sends *non-acknowledgement* (NACK) to the network indicating that a re-transmission of the same transport block is required. These data packet failures introduce latency to the network due to data re-transmissions. Such delays, caused by data failures, can have fatal consequences to applications that are sensitive to time latency, for example applications that require ultra reliable low-latency communication (URLLC, (Johansson et al., 2015). In order to cope with this latency, some naïve approaches have been adopted to solve the issue for URLLC, for example, using packet duplication (Rao and Vrzic, 2018) where the same transport block is transmitted more than once (usually twice or thrice) to the receiver. Several bursts of the same data packet ensure that at least one data packet will be successfully delivered to the receiver. All of the existing wireless networks either rely on re-transmitting the data packet upon the UE request when it fails to transmit or use packet duplication to avoid packet failures in advance when time latency is critical. To the best of author's knowledge, no intelligent solution exists that utilizes network data and measurements from radio network to solve this problem.

In wireless networks, transmission parameters are typically specified so that a given success rate is obtained. For instance, the system might be configured so that on average 1% of the data packets are lost due to noise or interference and must be re-transmitted. Allowing some of the transmissions to fail results in a higher spectral efficiency, at the cost of some additional latency and processing overhead. The main reason for not being able to guarantee 100% success rate for radio transmissions is the random nature of the wireless channels. If all foreseeable circumstances were to be covered, the data rates would be very low due to the redundancy required. In most of the cases, such extreme redundancy would not even be needed. Consequently, this randomness is best addressed by accepting that some of the transmissions fail and having a procedure for retransmitting them. However, some of the unpredictability of the wireless channel simply stems from the lack of a proper model for the underlying phenomena. In other words, by identifying these physical phenomena and modelling them by some means, it is possible to further improve the efficiency of wireless systems. For such problems, machine learning (ML, (Alpaydin, 2009)) is an excellent tool as it does not necessarily require one to know the model of the underlying

phenomenon. In this thesis, we aspire to design a solution that harnesses ML to make the wireless channel more predictable and thereby alleviates some of the randomness.

Therefore, the main problem that we aim to solve in this thesis is how to predict if the transmission of a given data packet is successful or not. This is a fundamental problem in wireless communication and solving this problem will facilitate higher spectral efficiency. It is clear that this type of a phenomenon cannot be perfectly predicted, but by combining several different measurements, some of the failures are likely to be predicted.

The problem we aim to solve is essentially a time series prediction problem. Network parameters that we use as features and labels are indexed over time and represent sequential data. Although traditionally, statistical forecasting methods like auto-regressive moving average (ARMA), auto-regressive integrated moving average (ARIMA), exponential smoothing (Hyndman and Athanasopoulos, 2019) have proven to be quite effective, recently, machine learning and neural networks have been remarkable in learning useful models using sequential data. From speech recognition/generation (Deng et al., 2013) to language translation (Sutskever et al., 2014), text summarization (Chuang and Yang, 2000), weather forecasting (Hong, 2008), product sales prediction (Sun et al., 2008) and so on, machine learning has been extremely useful in sequence modeling. In this thesis, we use state-of-the-art machine learning algorithms to predict data packet failures. For simplicity, one could imagine underlying problem to be a classification problem where classification has to be done in future with features and labels being time dependent.

The nature of available raw data and problem statement poses some challenges for the research goals. Main research challenges have been summarized below:

1. In a typical cellular network, ratio of failed packets (NACKs) to successful packets (ACKs) is quite low. For example, in the available dataset that we use, ratio of ACK to NACK is $15:1$, meaning that almost 93% of the time, data packets are delivered to the receivers without any errors. Since the aim is to predict data packet failures that happen almost 6% to 7% of the times, the desired class label is highly underrepresented and has significantly fewer training samples.

2. In the available dataset, there are 313 recorded network parameters (excluding time) that can be used to predict the packet failures. In principle, in machine learning, it is recommended to use only relevant features and exclude features that might not be helpful. This is because extraneous features can increase feature space, impede model learning, and increase the risk of over-fitting the training data (Kohavi and Sommerfield, 1995). In the dataset, narrowing down the list of parameters to the relevant parameters requires a lot of theoretical domain knowledge, experimentation, and analysis.

3. Data processing part, in particular, is quite challenging because of how users occupy timeslots in the cellular network. There are hundreds of unique users in data files who occupy timeslots in the network for an arbitrary amount of time in any order. Users can leave and occupy timeslots at any time in the network

(offline and online modes) which leaves gaps between users' time series data. Since future predictions depend on the past values, one needs to come up with a method for filling in the missing timeslots for each user and the corresponding labels. These dummy values must then be accounted for in model training.

Keeping these concerns and challenges in sight, we explore the feasibility of designing an intelligent solution for packet failure prediction based on the network data. Moreover, we also aim to determine the relevant radio parameters and number of optimal past values of these parameters for forecasting packets failures in the next transmission time interval (TTI).

2 Background

This chapter provides a brief overview of error handling in LTE and machine learning algorithms employed for model training.

2.1 Error Handling in LTE

In the history of network evolution, three different forces have consistently driven the architecture and evolution of telecommunications networks: traffic growth, development of new services, and advances in technology (El-Sayed and Jaffe, 2002). In the last four decades, the base stations in telecommunications have evolved from 2G to 3G, 4G and 5G. Base stations in 3G, called Node Bs (NB), are controlled by a central radio network, called Radio Network Controller (RNC). In 4G, base stations are no longer controlled by a central radio network and have individual control to schedule and dynamically allocate the resources to user equipment (UE), both in uplink and downlink. These independent base stations in 4G are called, evolved Node Bs (eNBs). In eNBs, resources are scheduled and allocated every 1 ms TTI (transmission time interval). So, eNBs algorithm runs every 1 ms and based on the radio and channel conditions, it allocates air interface resources to UEs that are connected to it. The user plane (U-plane), which carries network user traffic, protocol stack in LTE involves medium access layer (MAC). One of the important functionalities of MAC layer is error handling through hybrid automatic repeat request (HARQ).

HARQ is one of the core features that provides robustness in LTE (and advanced LTE networks) and is a hybrid of two error handling techniques: forward error correction (FEC) and automatic repeat request (ARQ) (Burton and Sullivan, 1972). It combines the important features of both ARQ and FEC error control. In ARQ, once the receiver receives the data packet, it performs cyclic redundancy check (CRC) (Enns and O'Hare, 1991) and sends the results to the transmitter. If the transmitter receives ACK, then the next packet is transmitted, whereas if NACK is received, then the transmitter re-transmits the data packet.

FEC is a method to enhance data reliability by detecting and correcting errors in the transmitted data (Davida and Reddy, 1972). In FEC, the sender sends a redundant error-correcting code along with the actual data frame. The receiver performs necessary checks on redundant bits and executes error-correcting code to generate the actual frame if it finds that the data is free from errors. It then removes the redundant bits before passing the message to the upper layers. The main limitation of FEC is that if there are too many errors, the data frames need to be re-transmitted.

2.2 Supervised Machine Learning

Machine learning algorithm where an input-target pair is provided for training is regarded as supervised machine learning method (Kotsiantis et al., 2007). Although traditionally classical statistical methods (e.g., ARMA, ARIMA) were widely used

for time series prediction, supervised machine learning methods have become really popular for sequence modeling nowadays. Any time series prediction problem can be transformed into supervised machine learning by shifting the labels forward in time.

$$y_{t+t_0} = f(X_t) \tag{1}$$

where X_t represents features at time t, $f(.)$ is some function and y_{t+t_0} is label shifted forward in time by t_0.

Concept illustrated in Equation (1) can be generalized for any supervised machine learning method. Therefore, Naive Bayes (Rish et al., 2001), Logistic Regression (Kleinbaum et al., 2002), Lasso Regression (Tibshirani, 1996), Random Decision Trees (Ho, 1995) and XGBoost (Chen and Guestrin, 2016) can be used for time series prediction. Among these methods, tree boosting is an effective and widely used supervised machine learning method.

2.2.1 XGBoost: Boosting Decision Trees

XGBoost, a scalable tree boosting system, was introduced by Chen and Guestrin (2016) and the algorithm has since been used in different classification and regression problems and Kaggle competitions. In the year 2015, among 29 winners of Kaggle competitions, 17 winners used XGBoost. Among these, 8 solely used XGBoost while the rest 11 used XGBoost with neural networks in ensembles (Chen and Guestrin, 2016). This shows the power of XGBoost which is quite simple to implement with open source library[1]. XGBoost works similar to boosted decision trees except that XGBoost model has been developed taking into consideration systems optimization and to push the limit of computational machine for scalability and parallel computations. The details can be found in the original paper (Chen and Guestrin, 2016).

Decision trees are one of the popular choices for supervised machine learning methods both for classification and regression (Vens et al., 2008; Chen and Liu, 2005). Knowledge learned by decision trees is preserved in the form of hierarchical trees where features represent the nodes and conditions on features are used for branching the trees. Figure 1 shows a simple example of a decision tree.

Decision trees are built using recursive binary tree splitting. In this recursive splitting, all features are considered and different splits are tried out. Splits with minimum cost function are kept. Trees end in leaf (node with no children) which decide the class of the input feature or some continuous value, depending on classification or regression problem. The building process of a decision tree can be considered two step: induction and pruning. During induction, features are represented as hierarchical structure and a tree is built. While pruning is applied to remove redundant structure from the trees. Pruning makes sure that decision tree does not overfit the training data and is able to generalize the predictions (Sheppard, 2017).

Even after pruning, decision trees tend to overfit and small changes in the dataset can lead to completely different trees generation, also known as variance. The bias and variance of decision trees can be mitigated using boosting and by training on multiple trees (ensemble technique) (Sheppard, 2017).

[1] https://github.com/dmlc/xgboost

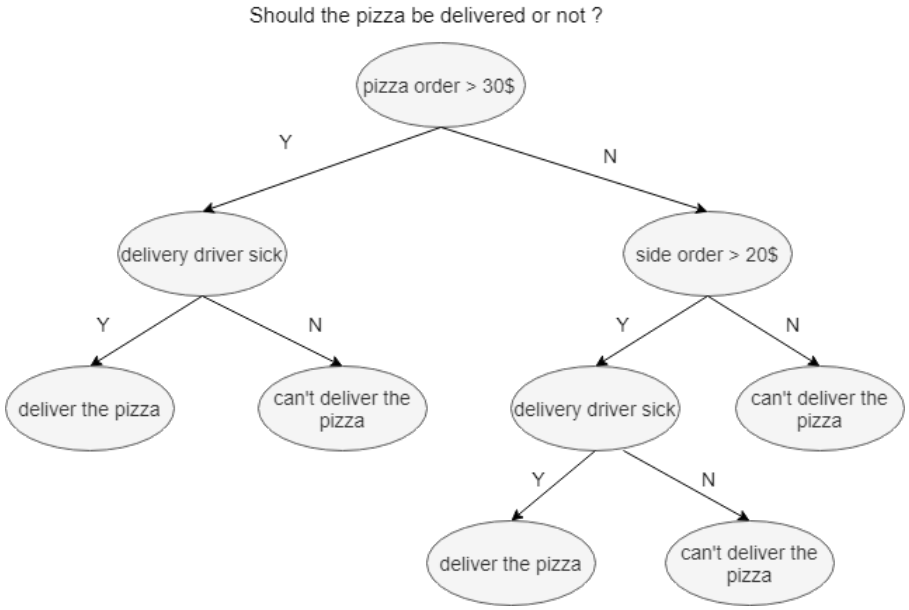


Figure 1: A simple example of a decision tree

Sometimes it is not sufficient to rely on just one machine learning method for reliable results, therefore, multiple learners/models are trained and their predictive power is combined into one aggregated output, like a voting system. Common ensemble methods are bagging and boosting (Bauer and Kohavi, 1999). Both of these ensemble methods use bootstrapping which is random selection of features with replacement. Bagging (abbreviation of bootstrap aggregation), aggregates the predictions of weak learners to predict the final output. Whereas in boosting, instead of simple averaging, weighted averaging is calculated using the predictions of the weak learners.

In XGBoost, trees are built sequentially and each subsequent tree aims to reduce the error of the previous tree. The base learners in XGBoost are weak learners with high bias. With weighted average of weak learner's predictions, a strong learner is obtained with low bias and variance (Sheppard, 2017).

Regression trees in XGBoost (also known as classification and regression trees, CART) are built over time and an iterative process is used where model tries to fix what it has learned and build a new tree. If we write predicted value at step t as $\hat{y_i}^{(t)}$ then

$$\begin{aligned}
\hat{y_i}^{(0)} &= 0 \\
\hat{y_i}^{(1)} &= f_1(x_i) = \hat{y_i}^{(0)} + f_1(x_i) \\
\hat{y_i}^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y_i}^{(1)} + f_2(x_i) \\
&\dots \\
\hat{y_i}^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y_i}^{(t-1)} + f_t(x_i)
\end{aligned} \tag{2}$$

where K is the total number of trees built, f_i is a function in the functional space

$\mathcal{F}$, the set of all possible CARTs. In other words, f_i are functions that contain the structure of the tree and the leaf scores.

As the additive training of trees continues and more trees are built, one needs to determine the trees to keep at each step. XGBoost keeps the trees that optimize the objective function. The general purpose equation for objective function in supervised learning is composed of a training loss and regularization term

$$O(\theta) = \mathcal{L}(\theta) + \Omega(\theta), \tag{3}$$

where $O(\theta)$ is the objective function, $\mathcal{L}(\theta)$ is the training loss function and $\Omega(\theta)$ denotes the regularization function.

Now the objective function for trees can be written as

$$\begin{aligned} O^{(t)} &= \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t)}) + \Omega(f_t), \\ O^{(t)} &= \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(x_i)) + \Omega(f_t), \end{aligned} \tag{4}$$

where l is a differentiable convex loss function that measures the difference between the prediction and the target.

The above Equation (4) involves functions as parameters and can not be optimized in Euclidean space as mentioned in the original paper by Chen and Guestrin (2016), therefore second order Taylor approximation is used to optimize the function:

$$O^{(t)} = \sum_{i=1}^{n} [l(y_i, \hat{y_i}^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i {f_t}^2(x_i)] + \Omega(f_t). \tag{5}$$

By dropping the constants:

$$O^{(t)} = \sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i {f_t}^2(x_i)] + \Omega(f_t), \tag{6}$$

where g_i and h_i are the first and second order derivative of the loss function.

Finally, the regularization term is given by:

$$\Omega(f_t) = \gamma T + \frac{1}{2} \lambda \|\omega\|^2, \tag{7}$$

where T is the number of leaves and w is the vector of scores on leaves. γ and λ are the hyperparameters. There are other choices of regularization function as well but above Equation 7 seems to work pretty well for practical problems (Chen and Guestrin, 2016).

When using XGBoost for binary classification, l (differentiable loss function) is the log likelihood of the Bernoulli distribution and expression can be summarized as follows:

$$l = y_i(\hat{y_i}^{(t-1)} + f_t(x_i)) - \log(1 + \exp(\hat{y_i}^{(t-1)} + f_t(x_i))). \tag{8}$$

2.3 Neural Networks

Neural networks are computing systems comprised of highly interconnected neurons capable of information processing due to their dynamic state response to external inputs (Goodfellow et al., 2016). Neural networks essentially mimic the biological nervous systems, which makes them great at solving problems without being explicitly programmed. Due to their ability to solve problems generally, neural networks have become the focus of prime research in 21st century. Neural networks have transformed human lives and have countless applications, for example in computer vision (Krizhevsky et al., 2012), natural language processing (Devlin et al., 2014; Lai et al., 2015) and health care (Lisboa and Taktak, 2006).

Although there are many different types of neural network models, the following section entails brief description of multilayer perceptron (MLP), long short-term memory (LSTM) and gated recurrent units (GRU) types of neural networks, since these are the only ones used in this thesis.

2.3.1 Multilayer Perceptron Model

Multilayer perceptron model is a network of simple perceptrons that were originally introduced by Rosenblatt (1958). A single layer perceptron network consists of weights w_i applied to inputs x_i and added with bias b followed by a non-linear activation function ϕ.

$$f(\boldsymbol{x}) = \phi(\sum_{i=0}^{n} w_i x_i + b) = \phi(\boldsymbol{w}^T \boldsymbol{x} + b) \tag{9}$$

where $\boldsymbol{w}$ is the weight vector, $\boldsymbol{x}$ is the vector of inputs.

A single layer perceptron network can be used to learn linear functions but cannot be used to perform complex tasks like learning a non-linear decision boundary in classification. On the other hand, a multilayer percetron network (MLP), which uses two or more layers of perceptrons, can be used to learn complex functions and non-linear decision boundaries (Kriesel, 2007). One way of training MLP neural networks is to use forward passes and backpropagation to learn the weights and bias. Forward passes (from input to output) calculate the outputs, while backpropagation calculates the error with respect to each weight and bias. The weights and biases are then updated accordingly.

Forward and backward passes in MLP can be better understood with an example. Figure 2 shows a simple MLP model with 2 hidden layers. In the figure, let w_{ij}^l be the weight that connects i-th neuron on $(l-1)$-th layer and j-th neuron on l-th layer, $\sigma(.)$ be the activation function for l-th layer, o_i^j be the output from the i-th neuron of l-th layer and O be the output of the neural network. One can show that using loss function $L(\boldsymbol{W}; \boldsymbol{x}, y) = \frac{1}{2}(O_{\boldsymbol{W},\boldsymbol{x}} - y)^2$, the forward pass equation for output can

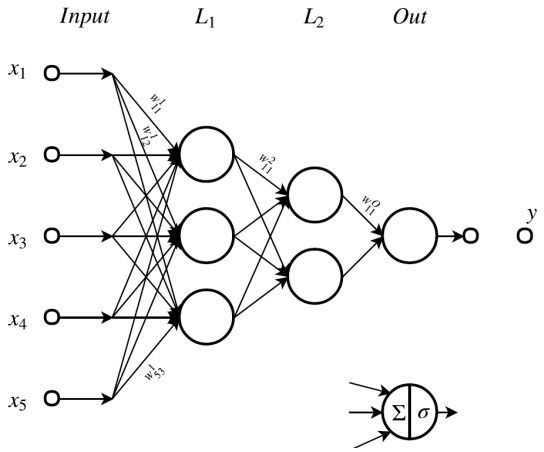


Figure 2: A simple MLP network with two hidden layers. Network receives five inputs and yields a single output.

be written as :

$$\begin{aligned} O_{\boldsymbol{W},\boldsymbol{x}} &= \sigma_3(o_1^2 w_{11}^3 + o_2^2 w_{21}^3 + b^3) \\ &= \sigma_3(\boldsymbol{o^2 w^3} + b_3) \\ &= \sigma_3(\Sigma_1^3) \end{aligned} \tag{10}$$

where $\Sigma_1^3 = \boldsymbol{o^2 w^3} + b_3$, $\boldsymbol{o^2} = [o_1^2, o_2^2]$ and $\boldsymbol{w}^3 = [w_{11}^3, w_{21}^3]^T$.

Similarly, forward pass equations for outputs of first and second neurons (o_1^2 and o_2^2) from layer 2 can be written as below:

$$\begin{aligned} o_1^2 &= \sigma_2(o_1^1 w_{11}^2 + o_2^1 w_{21}^2 + o_3^1 w_{31}^2 + b_1^2) \\ &= \sigma_2(\Sigma_1^2) \\ o_2^2 &= \sigma_2(o_1^1 w_{12}^2 + o_2^1 w_{22}^2 + o_3^1 w_{32}^2 + b_2^2) \\ &= \sigma_2(\Sigma_2^2) \end{aligned} \tag{11}$$

One could follow the same process, as described in Equation (10) and (11) to derive equations for forward passes for neurons in the first layer.

Backpropagation is at the heart of fine tuning an artificial neural network model where losses are backpropagated from outputs to inputs and by recursively applying derivative chain rule, gradients are computed. The concept can be better illustrated using Figure 2 and calculating backpropagation error equation for weight w_{11}^3.

$$\begin{aligned} \frac{\partial L(\boldsymbol{W};\boldsymbol{x},y)}{\partial w_{11}^3} &= \frac{\partial \frac{1}{2}(O_{\boldsymbol{W},\boldsymbol{x}} - y)^2}{\partial w_{11}^3} = (O_{\boldsymbol{W},\boldsymbol{x}} - y)\frac{O_{\boldsymbol{W},\boldsymbol{x}}}{\partial w_{11}^3} \\ &= \delta^o \frac{\partial \sigma_3(\Sigma_1^3)}{\partial \Sigma_1^3}\frac{\partial \Sigma_1^3}{\partial w_{11}^3} = \delta^o \sigma'(\Sigma_1^3) o_1^2 \\ &= \delta^3 o_1^2 \end{aligned} \tag{12}$$

where $\delta^o = (O_{\boldsymbol{W},\boldsymbol{x}} - y)$ is the output error, $\sigma'(.)$ denotes the derivative of sigmoid function with respect to its argument and $\delta^3 = \delta^o \sigma'(\Sigma_1^3)$.

Equation (12) calculates the error contribution towards the output because of the weight value w_{11}^3. Similarly one could write error equations for each weight following the same procedure. The weight update equation depends on the choice of optimizer. For instance with gradient descent, weight update equation for w_{11}^3 is as follows:

$$w_{11}^3 = w_{11}^3 - \alpha \times \frac{\partial L(\boldsymbol{W}; \boldsymbol{x}, y)}{\partial w_{11}^3} \tag{13}$$

where α is the learning rate of the optimizing algorithm.

Although MLP models are extensively used for binary and multiclass/multilabel classification problems, they have also proved to be effective for time series prediction (Tang and Fishwick, 1993; Haselsteiner and Pfurtscheller, 2000; Karunasinghe and Liong, 2006).

2.3.2 Recurrent Neural Networks

Feedforward neural networks do not have the capability to store any information since it has no feedback loop. Recurrent Neural Networks (RNNs), on the other hand, have loops in them, allowing information to persist (Goodfellow et al., 2016). Figure 3 shows the basic structure of a single RNN cell.

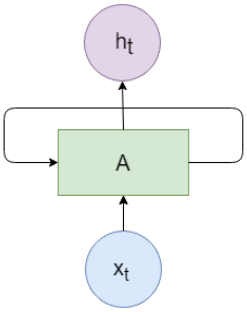


Figure 3: A single RNN Cell. The loop allows information to be passed onto the next step.

In Figure 3, x_t is the input at time t, A represents the basic cell structure of a particular RNN type, and h_t represents the output at time t. A recurrent network represents multiple copies of such a single RNN cell where each cell has the ability to store some information and pass it over to the next cell. More formally, given a sequence $\boldsymbol{x} = (\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_t)$, RNN updates its recurrent hidden state $\boldsymbol{h}_t$ at time t by:

$$\boldsymbol{h}_t = \begin{cases} 0, & \text{if } t = 0, \\ \phi(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t), & \text{otherwise,} \end{cases} \tag{14}$$

where ϕ is a nonlinear function.

Traditionally, the update of the recurrent hidden state in Equation (14) is implemented as follows:

$$\boldsymbol{h}_t = g(\boldsymbol{w}[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + b) \tag{15}$$

where $\boldsymbol{w}$ and b are weights and biases, g is a smooth, bounded function such as the logistic sigmoid function or hyperbolic tangent, and $[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t]$ denotes the concatenation of prediction $\boldsymbol{h}_{t-1}$ at previous time step and current input $\boldsymbol{x}_t$.

Because of the ability of RNNs to preserve information, RNNs have been quite popular in time series prediction and other sequential tasks like speech recognition, language translation (Devlin et al., 2014), and text classification (Lai et al., 2015). Below we discuss two popular RNN models: Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU).

2.3.3 Long Short-Term Memory (LSTM)

It was concluded by Bengio et al. (1994) that recurrent neural networks are not good at learning long term dependencies using gradient optimization methods because RNNs tend to suffer from vanishing (most of the time) and exploding (rarely but with severe effects) gradient problems (Pascanu et al., 2012), which make gradient optimization methods not useful for RNNs. In order to cope with this issue, LSTMs were introduced by Hochreiter and Schmidhuber (1997), which are a special type of RNNs and specialize in learning long-term dependencies.

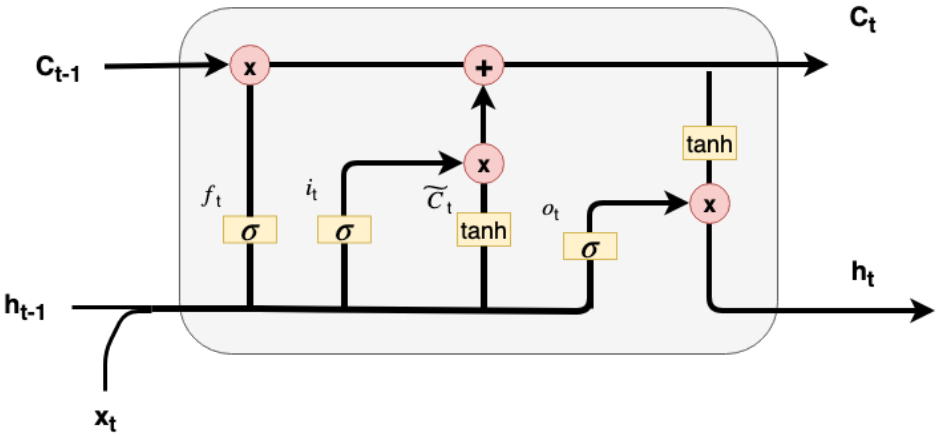


Figure 4: A single LSTM unit consists of three gates (input, forget and output) and a memory cell. Gates help LSTM cells to regulate flow of information and memory cell tries to memorize useful information.

Figure 4 shows the structure of a typical LSTM unit. A single LSTM unit consists of three gates and a memory cell. Gates act as regulators of information and help LSTM units to remove old information or add new information.

It is useful for LSTM units to keep updating their memories and forget any features that might not be useful anymore. The extent to which the existing memory

is forgotten is controlled by the *forget gate* ($\boldsymbol{f}_t$).

$$\boldsymbol{f}_t = \sigma(\boldsymbol{w}_f[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + b_f), \tag{16}$$

where $\boldsymbol{w}_f$ and b_f represent the weights and biases of the forget gate $\boldsymbol{f}_t$, and σ represents the sigmoid activation function.

After an LSTM unit has erased some part of its memory, the next step is to decide which new information should be added to the cell memory. Input gate decides how much updated values should be added to the cell memory $\tilde{\boldsymbol{C}}_t$ while tanh activation function calculates the updated values.

$$\begin{aligned} \boldsymbol{i}_t &= \sigma(\boldsymbol{w}_i[\boldsymbol{h_{t-1}}, \boldsymbol{x_t}] + b_i), \\ \tilde{\boldsymbol{C}}_t &= \tanh(\boldsymbol{w}_c[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + b_c), \end{aligned} \tag{17}$$

where $\boldsymbol{w}_i$ and b_i represent the weights and biases of the input gate $\boldsymbol{i}_t$ at time t, and $\boldsymbol{w}_c$ and b_c represent the weights and biases of cell memory $\tilde{\boldsymbol{C}}_t$ at time t.

Now that forget gate has decided to erase some memory content from the past and input gate has learned some useful information based on the new content, it is time to update the cell memory accordingly. The extent to which past memory is erased is controlled by the forget gate and the extent to which new content is added is controlled by the input gate:

$$\boldsymbol{C}_t = \boldsymbol{f}_t \odot \tilde{\boldsymbol{C}}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{C}}_t, \tag{18}$$

where $\odot$ represents the Hadamard product.

Finally, the output gate calculates the output and prediction at time t is calculated using tanh over current cell memory ($\tilde{\boldsymbol{C}}_t$) regulated by output gate $\boldsymbol{o}_t$.

$$\begin{aligned} \boldsymbol{o}_t &= \sigma(\boldsymbol{w}_o[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + b_o), \\ \boldsymbol{h}_t &= \boldsymbol{o}_t \tanh(\boldsymbol{C}_t). \end{aligned} \tag{19}$$

As compared to traditional RNNs where a cell consists of a logistic sigmoid or hyperbolic tangent, one can see from Figure 4 that a single cell in LSTM is more complicated and contains several gates and memory cell. Gates help LSTM cells to filter out relevant information and memory cell helps to memorize that information. So, if a cell detects an important feature, it will try to presist it thus capturing long-term dependencies.

2.3.4 Gated Recurrent Units (GRUs)

GRUs, similar to LSTMs, also solve the problem of vanishing gradient of traditional recurrent neural networks and they were introduced by Cho et al. (2014). As compared to LSTMs, where we have three gates (input, output, and forget gate), there are only two gates in GRUs, called reset and update gate; making GRUs simpler and faster than LSTMs. Figure (5) shows the structure of a typical unit in GRU.

The reset gate $\boldsymbol{r}_t^j$ of a jth GRU unit regulates whether or not to ignore the previous hidden state while update gate $\boldsymbol{z}_t^j$ decides whether the hidden state is to

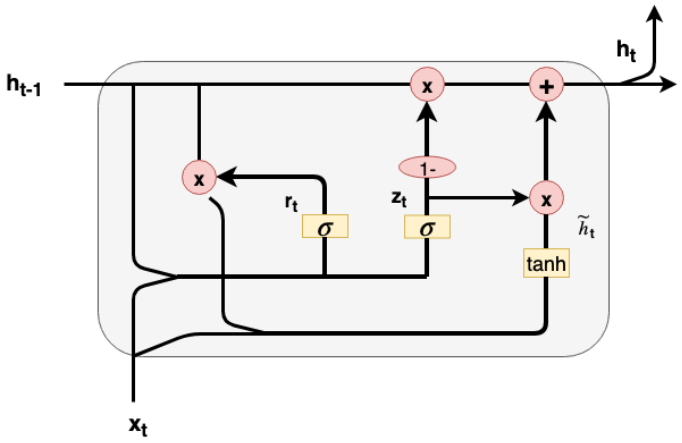


Figure 5: A single GRU unit consists of two gates, reset (r_t) and update gate (z_t) without having a separate memory cell as compared to LSTM.

be updated with a new hidden state $\tilde{\boldsymbol{h}}_t$. Mathematically both gates are calculated similary as follows:

$$\begin{aligned} \boldsymbol{r}_t^j &= \sigma(\boldsymbol{w}_r[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + b_r)^j, \\ \boldsymbol{z}_t^j &= \sigma(\boldsymbol{w}_z[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + b_z)^j. \end{aligned} \tag{20}$$

A new memory cell content $\tilde{\boldsymbol{h}}_t$ uses reset gate to preserve relevant information from the past.

$$\tilde{\boldsymbol{h}}_t^j = \tanh(\boldsymbol{w}[\boldsymbol{r}_t \odot \boldsymbol{h_{t-1}}, \boldsymbol{x}_t])^j \tag{21}$$

Final memory content $\boldsymbol{h}_t$ at time step t uses update gate to regulate information from the previous hidden state and the current new hidden state $\tilde{\boldsymbol{h}}_t$.

$$\boldsymbol{h}_t = (1 - \boldsymbol{z}_t) \odot \boldsymbol{h}_{t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{h}}_t. \tag{22}$$

Both LSTM and GRU model structures try to retain the past information, thus allowing information to persist. This can be seen as stacking of information with time with redundant or useless information being removed, allowing these models to learn useful temporaral structures within the data. Contrary to this, traditional RNNs replace the content of the previous hidden state by the new state, which inherently prevents them from learning any useful long-distance dependencies (Chung et al., 2014).

3 Materials and Methods

This chapter describes in detail the datasets used for model training, feature engineering, data processing, and visualization. Moreover, we also discuss the choice of machine learning libraries, baseline models, metrics for scoring model performance and available computational resources.

3.1 Dataset

For the purpose of thesis for research, dataset at hand has been provided by one of the clients of Nokia, China Mobile Telecommunication Company (CMCC), in comma-separated values (CSV). Several datafiles are at disposal where each file consists of approximately 0.1 million datapoints and 314 columns/features including time. Data inside the files represents the network data being utilized by different users for a particular cellular base station. Features represent the values for different network parameters. For this particular problem, domain expert knowledge suggests that not all parameters are useful to predict packet failures. Therefore, with the help of domain experts, 11 most relevant network parameters were selected out of total 314 parameters. These are the parameters that were used subsequently in data processing and model training. Brief description of each parameter is given below:

1. **ETtiTraceDlParUe_crnti** is the cell radio network temporary identifier (C-RNTI) of the user equipment (UE) allocated by the base station (BS). C-RNTI is unique within one cell controlled by the BS and can be reallocated when a UE moves to a new cell. This is essentially the ID of the UE.

2. **ETtiTraceDlParUe_wbCqiCompensateCw0** represents the channel quality indicator (CQI) (Love et al., 2008) of the network, meaning how strong the received signal is compared to the noise and interference.

3. **ETtiTraceDlParUe_rrmDeltaCqiCw0** is the correction factor to the CQI, which is different for each UE and is changed in accordance with the observed performance. This is needed because of the inherent biases in measuring the CQI.

4. **ETtiTraceDlParUe_rrmMimoCqi** represents the combined CQI for both of the received signals since the UE is receiving two signals at the same time (multi-antenna transmissions).

5. **ETtiTraceDlParUe_ModulationCw1** represents the modulation scheme (Proakis, 2001) being used in the signal.

6. **ETtiTraceDlParUe_mcsIndexCw1** represents a number corresponding to the modulation (in this case represented by ETtiTraceDlParUe_ModulationCw1) and coding rate, the latter of which defines how much redundancy there is in the raw data.

7. **ETtiTraceDlParUe_averCirRgbCw0** provides information as to how strong the signal is compared to interference. Basically, this is the average carrier-to-interference ratio, averaged over radio bearer groups (RBGs) (Ku, 2011).

8. **ETtiTraceDlParUe_dlrrmPdcchCqiShift** represents the correction applied to the CQI for the control data (physical downlink control channel, PDCCH).

9. **ETtiTraceDlParUe_rrmPdschAvgUeTput** is the average downlink (link from base station to the receiver) throughput.

10. **ETtiTraceDlParUe_rrmPdschAvgResAllocationUe** is the average amount of resources allocated for the UE, that is providing information about the bandwidth the UE is allowed to use.

11. **EHarqParDl_rrmRecommendedMcsCw1** is the recommended modulation and coding scheme for the possible retransmission of data packet in case of transmission failure.

To train the models, two types of datasets have been used. *Single datafile* consists of a single CSV file while *combined dataset* consists of 10 single datafiles. The idea is to start simpler with one CSV file and then combine multiple datafiles to see how model performance changes with bigger datasets. Python[2] (version: 3.6.7) has been used as the scripting language. Although there are many good frameworks available for implementing machine learning algorithms in Python, Keras[3] stands out because it is convenient to use and facilitates quick prototyping. Keras functional API provides an easy interface to implement RNNs and MLP in Python. For Python implementation of XGBoost, official documentation[4] provides details on Python API references including Scikit-Learn API. Since our problem deals with binary classification, Scikit-Learn API for XGBoost classification was used. Lastly, for computational resources, NVIDIA® Tesla® V100[5] was used which is a data center GPU built to accelerate AI, high performance computing (HPC), data science, and graphics.

3.1.1 Data Preprocessing

In order to prepare data for training, it is necessary to convert raw data into useful structured dataset. Data preprocessing in our problem consists of two major tasks:

1. Extracting data for a particular UE.

[2]https://www.python.org/
[3]https://keras.io/
[4]https://xgboost.readthedocs.io
[5]https://www.nvidia.com/en-us/data-center/tesla-v100/

2. Converting UE data to equal resolution (1 ms) by using filltype *zeros* or *previous*. The filltype *zeros* means that 0s will be used for missing timestamps, while filltype *previous* means that parameter value found in the last known timestamp will be used for missing timestamps.

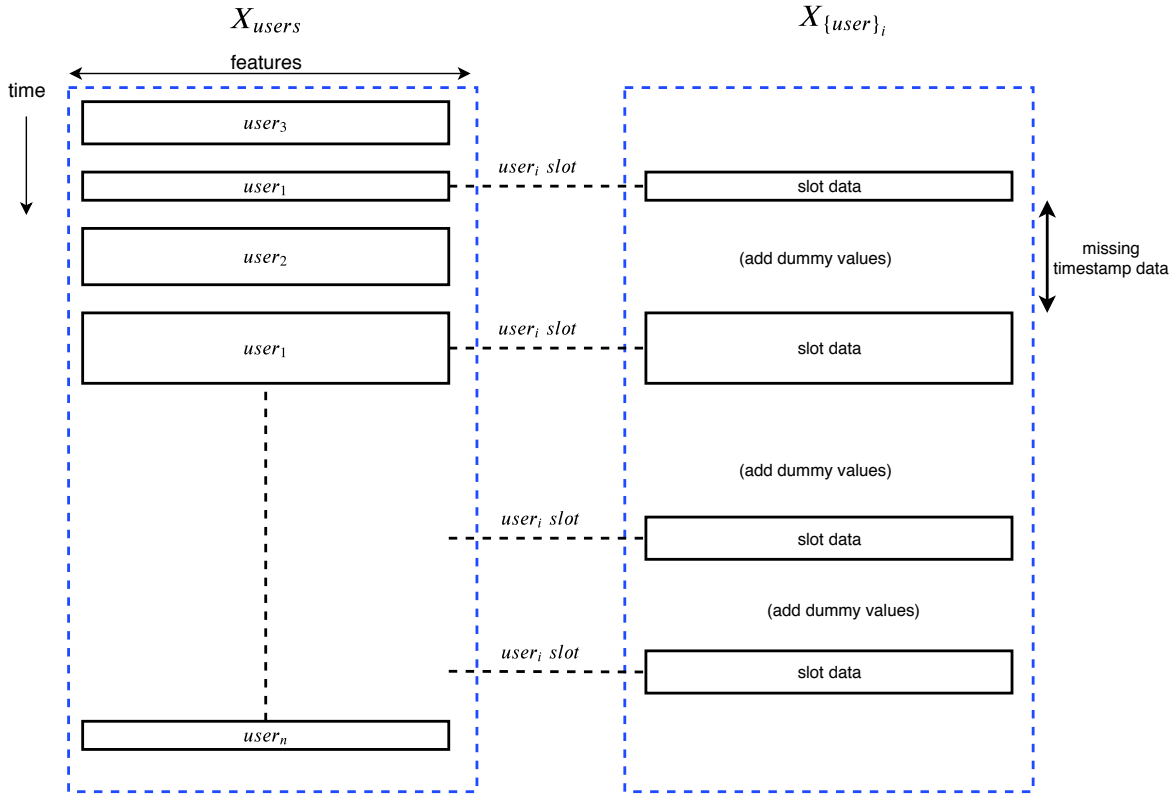


Figure 6: Data preprocessing for users. Each file contains several users. Users' data is extracted separately, which leaves empty gaps between their timestamps. These empty gaps are filled either using zeros or previous value as shown in Table 1.

Figure 6 shows data preprocessing more clearly. One can observe that users occupy timeslots for variable amounts of time, that is, occupied timeslots of the users are different. Moreover, the order in which users occupy timeslots does not follow any deterministic pattern. Since data of a user has missing timestamps (a user can go offline/online depending on the usage), it is necessary to fill certain values to the missing timestamps depending on the parameter. Table 1 shows how values are filled for missing timestamps. With filled values, we obtain user data X_{user_i} which is equally sampled at 1 ms frequency. Same process can be repeated for n different users.

Theoretically, data of a user is independent from other users. Therefore, it is necessary to extract the data for each individual user and then use the past data values of that particular user to predict whether a packet failure would happen for the user. In order to extract the data for a particular UE, one can use *ETtiTraceDlParUe_crnti* parameter.

Table 1: Scheme for filling missed timestamps. Second column in table shows which filltype has been used for the corresponding parameter in first column.

Parameter	Filltype
ETtiTraceDlParUe_wbCqiCompensateCw0	previous
ETtiTraceDlParUe_rrmDeltaCqiCw0	previous
ETtiTraceDlParUe_rrmMimoCqi	previous
ETtiTraceDlParUe_mcsIndexCw1	previous
ETtiTraceDlParUe_ModulationCw1	previous
ETtiTraceDlParUe_averCirRgbCw0	previous
ETtiTraceDlParUe_dlrrmPdcchCqiShift	previous
ETtiTraceDlParUe_rrmPdschAvgUeTput	zeros
ETtiTraceDlParUe_rrmPdschAvgResAllocationUe	zeros
EHarqParDl_rrmRecommendedMcsCw1	previous

3.1.2 Feature Engineering

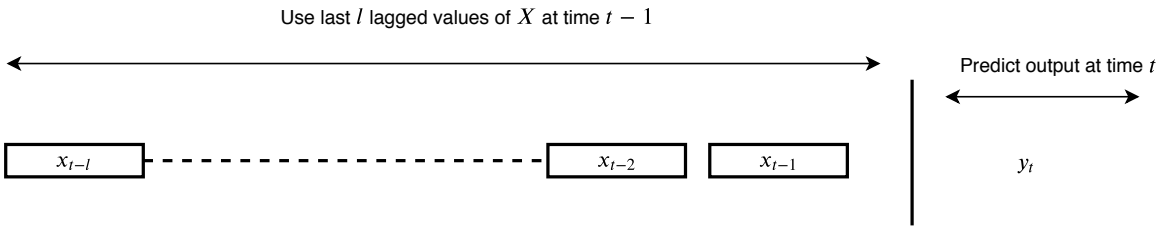


Figure 7: Feature engineering for time series prediction. Last l values of each feature are used to predict next output.

In order to prepare features for time series prediction, it is a common practice to use lagged values of both features and labels to predict the future values of features and labels or just labels. In our case, we only use the lagged values of features to predict the labels as per Equation (1). This can be understood as shifting the labels by some constant timestamp t_l and training the model to predict future values of labels given features at current time t, as shown in Figure 7. A good question to ask at this point is to how many past values of features should the model use to (referred commonly as *past window size*) to predict how much in the future (commonly named as *future window size*). In most of the applications, future window size is usually an application requirement, rather than a value to be determined, whereas past windows size needs to be experimented upon and chosen according to the best results (more discussion on this will follow in Section 4.4).

3.1.3 Data Visualization

For visualization in this part, we will focus on one CSV file (details about the file will follow in Section 4.1). In one CSV file, there can be around 1000 different users.

Each user can have different packet drop rate which depends upon different factors like location of user, and distance from the base station. Since data packets drop at different rates for different users, in order to provide a comprehensive understanding of packets drop rates at extreme ends, one can find ACK/NACK plot graphs in Figures 8 and 9 for two types of users: user u_1 who had the most dropped packets, and user u_2 who had the least amount of packets dropped. Please note that green dots represent ACK while red dots represents NACK.

With higher dimensional data, it can be difficult to visualize the data. In order to cope with this, dimensionality reduction techniques are used. With reduced dimensions one can easily plot the relation between X and y and infer useful information about the data. Below list entails the dimensionality reduction techniques being employed for data visualization. Please note that dimensionality methods use past window size of 15 for feature engineering, and future window size of 1 for labels.

1. **Principal Component Analysis (PCA)**

   PCA is one of the most common methods used for dimensionality reduction. PCA tries to remove redundant information by finding the most relevant linear combinations of variables. It finds the principal components (eigenvectors of covariance matrix) of the original dataset. These components correspond to the direction with greatest variance in dataset and are orthogonal to each other (Jolliffe, 2011)

2. **Independent Component Analysis (ICA)**

   As compared to PCA, where basis explain the variability of data, in ICA we aim to find basis that construct vectors which are independent components of the data. These independent components are assumed to be non-Gaussian, mutually independent and their linear combinations tend to explain variables of the data. As compared to PCA, where vectors are orthogonal, vectors in ICA are not orthogonal (Langlois et al., 2010).

3. **T-distributed Stochastic Neighbor Embedding (t-SNE)**

   t-SNE is a non-linear dimensionality reduction method, introduced by der Maaten and Hinton (2008). t-SNE minimizes the Kullback-Leibler divergence (Kullback and Leibler, 1951) of two probability distributions: a distribution that measures pairwise similarity of input data points in the higher dimensional space and the other distribution that measures similarities of embedded points in the correspondig low dimensional space.

Figures 10, 11 and 12 show the graph for reduced dimensions of original dataset using PCA, ICA, and t-SNE methods respectively. One can observe from PCA/ICA graphs that it is difficult to get a clear decision boundary to separate labels into two groups indicating that problem at hand overlaps the labels for given features of dataset and a linear model will not be sufficient to achieve a good decision boundary. Moreover, with non-linear dimensionality reduction method, t-SNE, we still do not observe two distinct clusters. This indicates that problem at hand is highly non-linear and a complex model is required to achieve a good binary classifier.

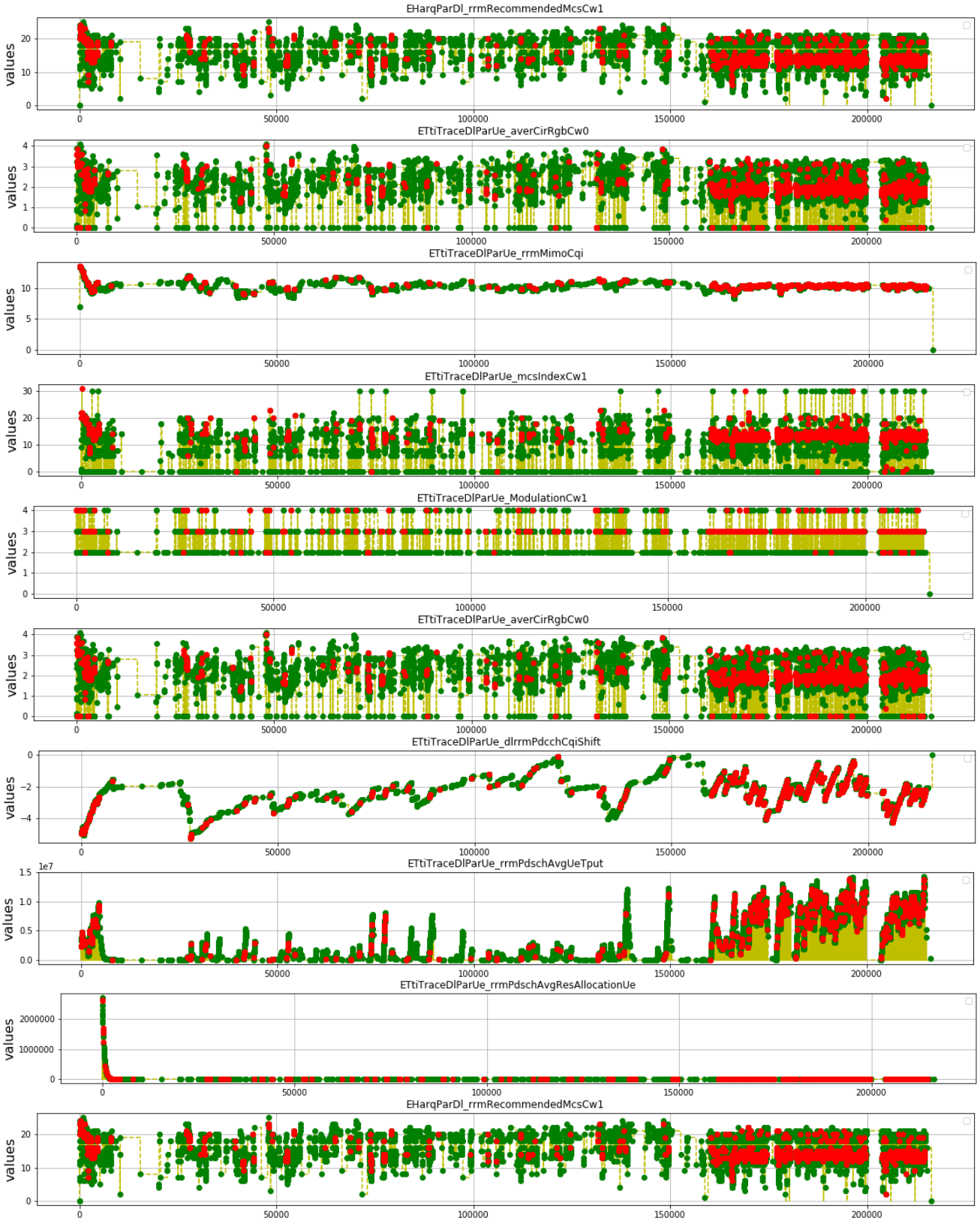


Figure 8: User with most dropped packets. Dot plot graphs show the distribution of labels against each feature (or radio parameter).

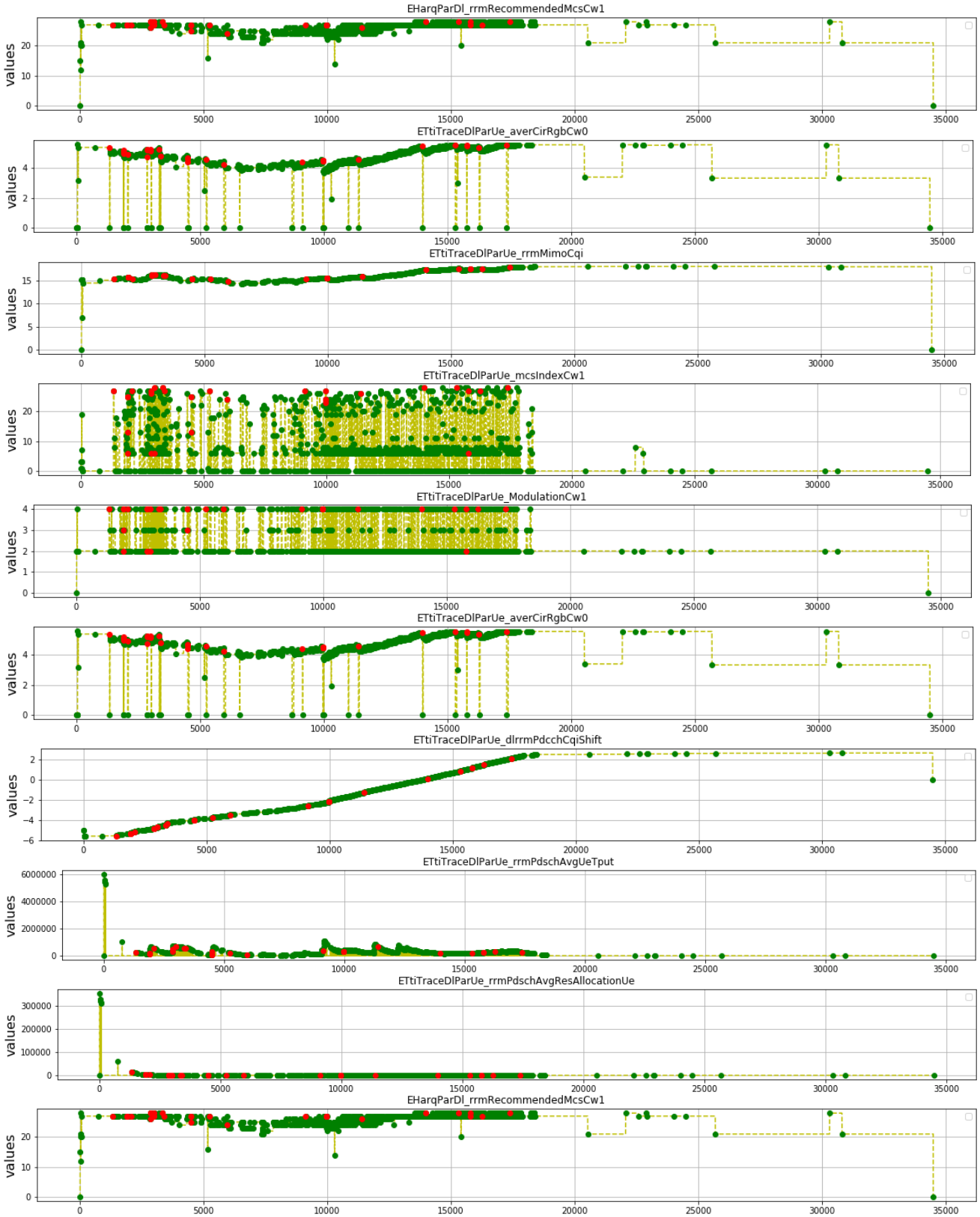


Figure 9: User with least dropped packets. Dot plot graphs show the distribution of labels against each feature (or radio parameter).

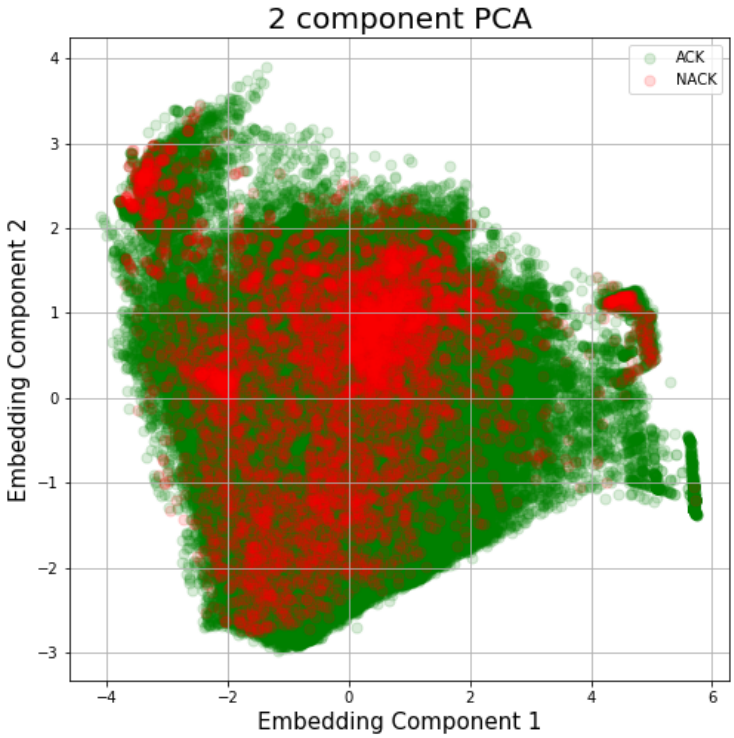


Figure 10: Dimensionality reduction using PCA.

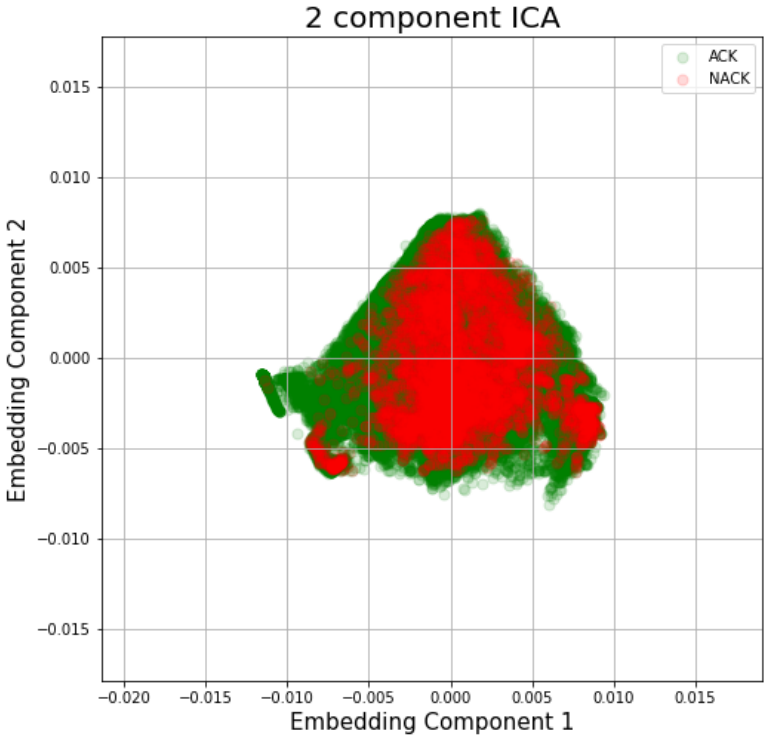


Figure 11: Dimensionality reduction using ICA.

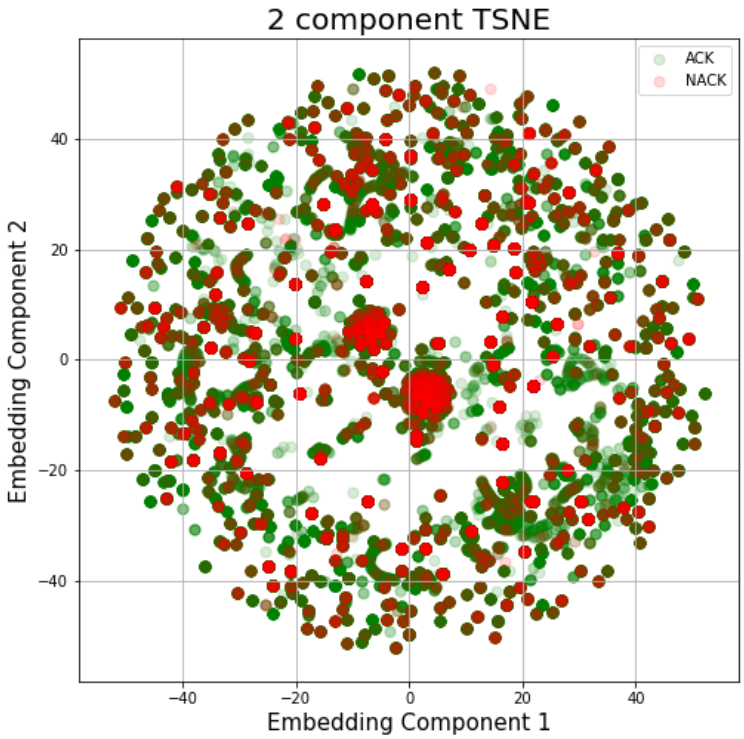


Figure 12: Dimensionality reduction using t-SNE.

3.2 Choosing the Right Metric

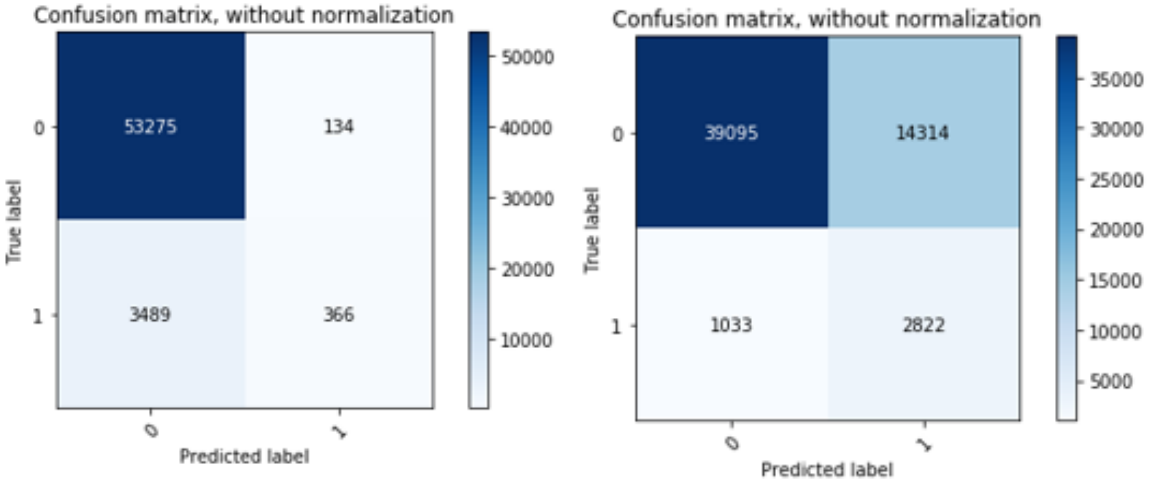
Metric in machine learning is defined as the measure of model performance. In order to evaluate the performance of a model, it is imperative to choose the right metric depending on the objective and dataset. Choice of the right metric can help to realize the improvement in model training and identify best hyperparameters. For example, choosing a mean squared prediction error (MSPE) or mean squared abolute error (MSAE) to gauge the performance of a binary classifier is not helpful since MSPE and MSAE are more helpful in regression; on the other hand, cross-entropy (Goodfellow et al., 2016) is a useful metric in classification.

For our dataset, since the classes are highly imbalanced, it is necessary to use a metric that takes into account class imbalance. Python Scikit-learn library[6] provides useful metrics for imbalanced classes. F1 score (Powers, 2011) is one of the most popular choices for imbalanced datasets. Using F1 score for our dataset, changes in accuracy of the trained model did not affect F1 score sufficiently making it difficult to point out the improvement in model training. Among the alternate options that were considered, balanced accuracy score (Brodersen et al., 2010), defined as the average of recall obtained on each class, and area under the (Receiver Operating Characteristics) curve (AUC) score (Fawcett, 2006) are better options.

In order to compare the performance of models, AUC is used as the decisive metric. This is because balanced accuracy score can differ depending on the probability

[6]https://scikit-learn.org/

threshold. This point can be best illustrated using Figures 13a and 13b. Confusion matrix in Figure 13a shows predictions against a probability threshold of 0.50; meaning if a data packet failure probability is equal to or greater than 50%, only then it will be classified as NACK. Similar interpretation follows for confusion matrix in Figure 13b with 0.07 probability threshold. Balanced accuracy score for both of these confusion matrices evaluates to 54.20 and 73.20 respectively; but area under the ROC curve is 0.82 for both confusion matrices. This is because balance accuracy score can vary depending on the probability threshold while AUC score tells the complete story regardless of what threshold is defined. AUC score varies from 0.50 to 1.0 where a higher score indicates a better classifier. In the following sections, balanced accuracy score will be mentioned as an extra insight for predictions with probability threshold of 0.50.



(a) (b)

Figure 13: Confusion matrix with different probability thresholds. Left Figure (a) shows confusion matrix with 0.50 probability threshold while right Figure (b) shows confusion matrix with 0.07 probability threshold.

3.3 Baseline Models

In order to gauge how well a trained model is performing, a baseline model accuracy can help to set a benchmark. This benchmark can be quite useful in determining how well a model is performing as compared to a dummy/non-trained model. This also highlights the potential benefit of using extra hardware computations and sophisticated algorithms for higher performance. If trained model's accuracy is close to or below the benchmark, there is no use of utilizing hardware resources for model training and evaluation.

In time series prediction, persistence model (where predicted value at time $t+1$ is same as the actual value at time t) is a common choice for benchmarking the trained model for a continuous time series. Since in our case, labels are categorical and classes are highly imabalanced, *zero rule algorithm*, where dominant class is

always predicted by the dummy model, is a better choice. Table 2 shows the baseline performance scores for both datasets (single and combined dataset).

Table 2: Benchmark scores for single and combined datasets.

Metric	**Benchmark score**
Balanced accuracy score	50.00 %
AUC	0.50

4 Results

In this chapter, we describe in detail the results of using different machine learning models on the available dataset and benchmark their performances according to AUC metric. Furthermore, we also show the results for feature selection and using different window sizes (past values) of features. For model training and evaluation, following machine learning algorithms are used: long short-term memory (LSTM), gated recurrent units (GRU), multilayer perceptron model (MLP), and boosting decision trees (XGBoost). For all neural networks, categorical cross entropy (Goodfellow et al., 2016) is used as the loss function since it has been proven to be quite effective in classification problems (Anthimopoulos et al., 2016). Please note that for all the experiments on both single datafile and combined dataset, past window size of 15 is used to predict next 1 ms unless stated otherwise.

4.1 Using Single Datafile

Initially, only one datafile (or one CSV file) with total 90,893 samples was used to experiment with different models and evaluate their results and comparative performance. For this datafile, Table 3 and 4 show class labels weight and data split ratio for training, validation, and testing respectively. Please note that *ACK* and *NACK* class labels correspond to class 0 and class 1 respectively.

Table 3: Class labels weight summary for single datafile. It can be seen that classes are highly imbalanced and dataset is skewed.

Class label	**Weight**
class 0 (ACK)	93.45 %
class 1 (NACK)	6.55 %

Table 4: Data split ratio for training, validation, and testing. 80% data is used for training while the rest 20% is used for validation and testing.

Dataset	**Split ratio(%)**
Training data	80.00
Validation data	12.00
Testing data	8.00

Next we will enlist the result of using each machine learning model on this dataset in Sections 4.1.1, 4.1.2, 4.1.3, and 4.1.4.

4.1.1 Long Short-Term Memory (LSTM)

Table 5 shows the hyperparameter configuration and their results using single datafile with LSTM. Below description includes explanation of parameters included in Table

5. Similar description also follows for tables summarizing results for GRU and MLP in Sections 4.1.2 and 4.1.3.

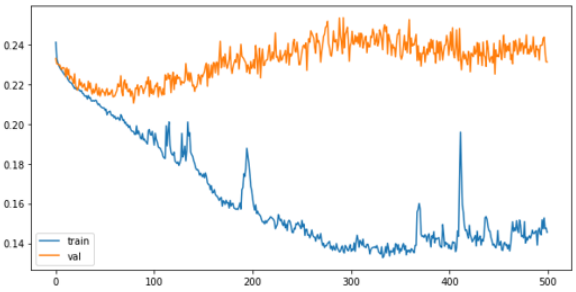
- **LSTM/GRU Units**: represents the dimensionality of hidden state (same as output state).
- **LSTM/GRU Layers**: total number of stacked layers used in LSTM/GRU. Please note that each stacked layer has same number of LSTM/GRU units.
- **Epochs**: total number of forward and backward passes completed in the neural network for all training examples.
- **ES** : stands for early stopping; this is a useful criterion that can be used to avoid overfitting of the model (Prechelt, 1998). A positive integer of ES represents the number of epochs after which further training must be stopped if model is not learning anymore.
- **BS**: stands for batch size; number of samples that are propagated through the neural network.
- **CW**: stands for class weight; this parameter can be useful while training on imbalanced dataset. More weight can be given to an under-represented class to force model to penalize its false detection. The parameter is calculated as $total\_samples/(n\_classes * count(class_i))$, where count represents the total number of occurrence of a class i in the dataset.
- **Dropouts**: a regularization technique where certain number of units in a neural network are dropped. Please note that in results of the following sections, dropouts in LSTM and GRU refer to *recurrent dropouts*.
- **Activation**: represents the non-linear activation function used for LSTM layer. For dense layer in LSTM and GRU, softmax activation is used.
- **Optimizer**: function used to learn best parameters (weights) of the model by reducing given loss function.
- **lr**: stands for learning rate of an optimizer, that is how big/small steps an optimizer needs to take in order to update model parameters (weights).
- **Model Parameters**: total number of trainable parameters in the model.
- **Balanced Acc**: balanced accuracy metric to score model performance.
- **AUC**: represents the area under the ROC curve score.
- **Time(h)**: time, in hours, taken for model to train.
- **Figures**: reference to loss history and confusion matrix figures.

Table 5: LSTM results summary on single datafile.

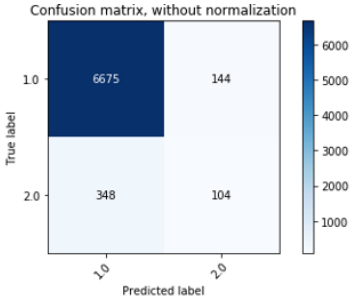
No.	LSTM Units	LSTM Layers	Epochs	ES	BS	CW	Dropout	Optimizer	lr	Model Parameters	Balanced Acc	AUC	Time(h)	Figures
1	200	1	500	NA	96	False	NA	RMSProp	0.001	168,800	68.49	0.77	3.9	14a, 14b
2	150	1	300	100	96	True	NA	RMSProp	0.001	966,00	55.77	0.75	1.42	14c, 14d
3	150	1	300	100	150	True	NA	RMSProp	0.01	966,00	0.50	0.50	0.5	14e, 14f
4	121	1	500	70	15	True	NA	Adam	0.001	261,000	53.38	0.72	6.09	14g, 14h
5	80	2	210	70	96	False	0.3	Adam	0.001	29,120	63.09	0.84	3	15a, 15b
6	80	2	448	70	96	True	0.3	Adam	0.001	29,120	60.33	0.82	6.3	15c, 15d
7	80	3	350	70	96	True	0.25	Adam	0.003	132,322	62.41	0.84	7.7	16a, 16b

One can observe from the loss history of Figures 14a, 14c, 14e, and 14g that without using dropouts in LSTM, model predominantly overfits the training data and validation loss starts increasing after certain number of epochs. Moreover, from Figures 14b, 14d, 14f, and 14h of confusion matrix, one can see that not a lot of failed packets are predicted in the next TTI. This results in lower AUC score, indicating that model does not learn well. In order to overcome this problem, number of LSTM layers were increased; usually multilayered LSTMs are able to give better results for complex tasks. One can see from experiments number 5, 6 and 7 in Table 5 that increasing the number of stacked layers and reducing LSTM units helps with better performance. It is always a good idea not to use too many LSTM units and LSTM layers since model can become too complex resulting in overfitting and increased training time. In order to expedite the training time and tune the model better, different number of batch sizes were also tried out. Using 96 samples for batch size gives better results because if batch size is increased further, model performance decreases. Among different optimizers that were tried out, *Adam* (Kingma and Ba, 2014) optimizer comes out to be the best choice. Figures 15a, 15b, 16a, and 16b show the loss history and confusion matrix for the best results with AUC score of 0.84.

One can observe from the results in Table 5 that recurrent dropouts help with regularization and to close down the gap between validation and training loss. Moreover, one can see that stacked LSTMs perform better than single layer LSTM, reason being that stacked LSTMs can learn more complex functions. A common understanding in deep learning is that deep RNNs work better than shallower ones on some tasks (Goldberg, 2016).

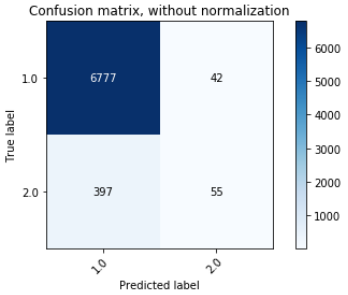


(a) Loss history for experiment 1

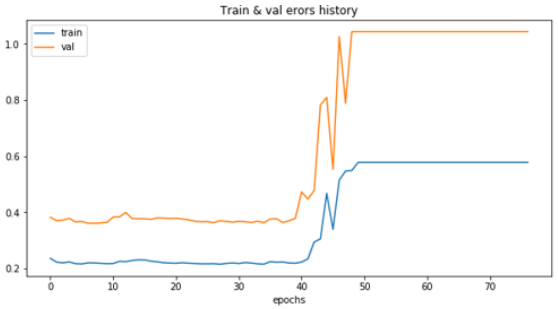


(b) Confusion matrix for experiment 1

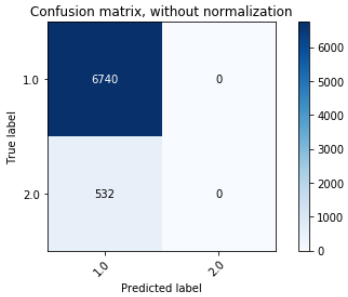
(c) Loss history for experiment 2



(d) Confusion matrix for experiment 2

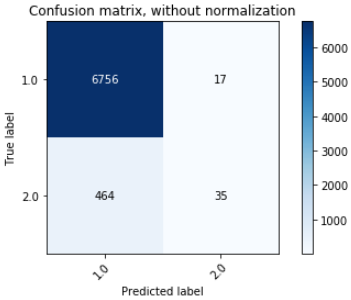


(e) Loss history for experiment 3



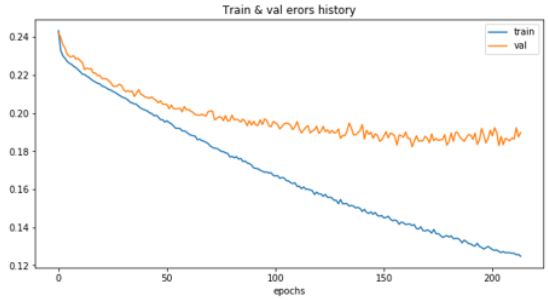
(f) Confusion matrix for experiment 3

(g) Loss history for experiment 4

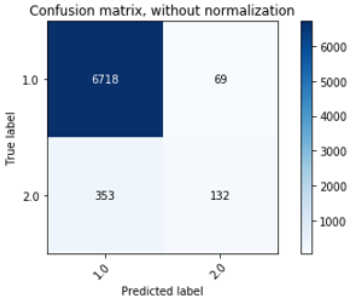


(h) Confusion matrix for experiment 4

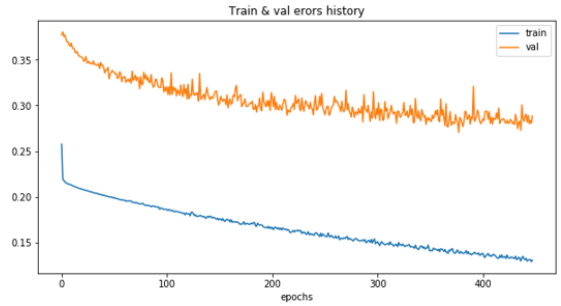
Figure 14: Training and validation loss history for single layer LSTMs. Without using dropouts, LSTM overfits the training data and validation loss increases after some epochs.



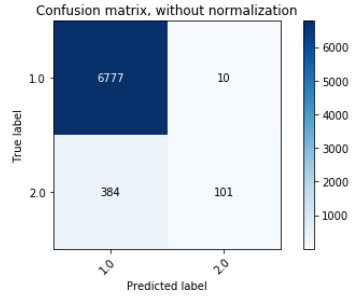
(a) Loss history for experiment 1



(b) Confusion matrix for experiment 1



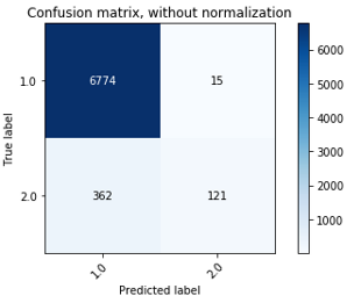
(c) Loss history for experiment 2



(d) Confusion matrix for experiment 2

Figure 15: Loss history and confusion matrix for two layer stacked LSTMs. Using two layered LSTMs helps with better performance. Dropouts and early stopping are used to make sure that model does not overfit.

(a) Loss history for experiment 1



(b) Confusion matrix for experiment 1

Figure 16: Loss history and confusion matrix for three layer stacked LSTMs. Three layered LSTMs give comparable performance as compared to two layered LSTMs except training time increases significantly.

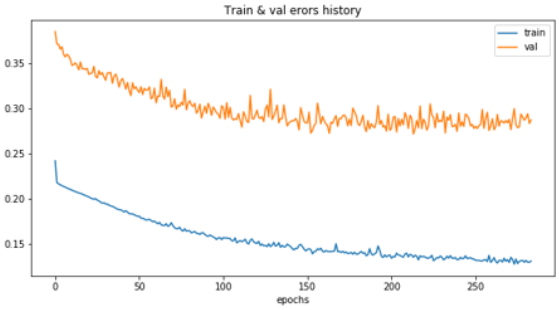
4.1.2 Gated Recurrent Units (GRUs)

As mentioned earlier in chapter 2, GRUs are really useful in sequential tasks like LSTMs, but only faster and simpler. Table 6 summarizes the best results obtained with GRUs in Keras using single datafile.

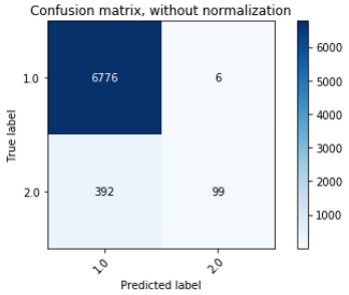
Table 6: GRU results summary on single datafile

No.	GRU Units	GRU Layers	Epochs	ES	BS	CW	Recurrent dropouts	Optimizer	lr	Model Parameters	Balanced Acc.	AUC	Time(h)	Figures
1	150	2	284	70	96	True	0.25	Adam	0.001	208202	60.03	0.86	3.36	17a, 17b
2	100	4	125	70	200	False	0.3	Adam	0.001	33502	50.48	0.70	0.32	18a, 18b

Using hyperparmeter values in experiment 1 in Table 6, an AUC score of 0.86 is obtained which is slightly better than the best result obtained with LSTMs. Figure 17a shows that both training and validation loss decrease slowly as the epochs increase. From Figure 17b, one can see that 99 packets failures are predicted successfully with total 6 false predictions of failed packets. On the contrary, experiment 2 in Table 6 shows poor GRU performance using the mentioned hyperparameters. Although training loss decreases, validation loss starts increasing as shown in Figure 18a. Confusion matrix in Figure 18b shows that only 5 NACKs are predicted successfully out of almost 500 total NACKs.

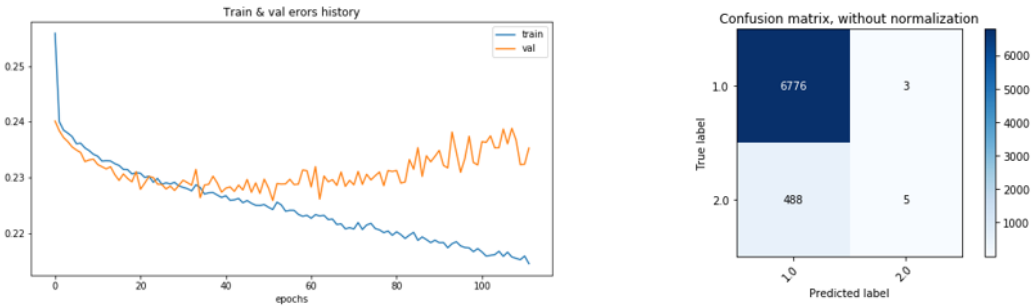


(a) Loss history for experiment 1



(b) Confusion matrix for experiment 1

Figure 17: Loss history and confusion matrix for two layer stacked GRUs. Using simple two layered GRU gives even better results than LSTMs.



(a) Loss history for experiment 1 (b) Confusion matrix for experiment 1

Figure 18: Loss history and confusion matrix for four layer stacked GRUs. Using too many stacked layers overfits the data and model does not learn as can be seen from both left (a) and right (b) figures.

4.1.3 Multilayer Perceptron (MLP)

Table 7 shows the results obtained with MLP with two hidden layers using single datafile. For each of the experiment listed in Table 7, one can find corresponding loss history for training and validation data in Figures 19a, 19c, 19e, 19g, 19i, 19k, and 19m. As can be seen from Figure 19a, using ReLU (Glorot et al., 2011) activation, validation data loss starts to increase after 50 epochs. Compared to this, using *tanh* activation, validation loss decreases smoothly over number of epochs and also follows training loss closely. Moreover, using *tanh* activation, slightly better AUC score is obtained as compared to ReLU activation. Therefore, in the rest of the experiments, *tanh* activation function was used.

From the experiments in Table 7, one can observe that increasing number of neurons in hidden layers slightly helps with better performance. In experiment 3, by doubling the number of neurons as compared to experiment 2, better model performance is achieved, with 0.02 increase in AUC score. Similarly, in experiment 7, one can observe that if the number of neurons is decreased as compared to experiment 6, then AUC score decreases slightly (although the difference is not noticeable). Experiment 4 in Table 7 lists the hyperparameters that gives the best results with 2 hidden layered MLP using 512 and 256 neurons in the first and second layer respectively. While tuning number of layers and neurons in MLP, it is always preferable to get good model accuracy using a simpler model since a complex model will learn the training data too well and will not be able to deal with the unseen dataset. Therefore, it is always preferable to use a simple model; and if simple model does not work then one needs to find a balance between a slightly complex model that barely gets the job done and a complicated model that might lead to overfitting. While tuning number of neurons in Table 7, this principle was strictly adhered to.

Moreover, among different batch sizes that were tried out, using a batch size of 256 gives better results. If batch size is increased more, for instance 400, model performance drops (although model takes less time to train). Lastly, using Adamax optimizer (Kingma and Ba, 2014), a variant of Adam based on the infinity norm, model performance increases.

Table 7: MLP with two hidden layers results summary on single datafile.

No.	H_1 units	H_2 units	Epochs	ES	BS	Activation	Optimizer	lr	Model Parameters	Balanced Acc.	AUC	Time(h)	Figures
1	512	256	214	150	256	relu	Adam	0.0009	209,154	51.73	0.78	0.10	19a, 19b
2	512	256	398	150	256	tanh	Adam	0.0009	209,154	53.60	0.79	0.25	19c, 19d
3	1024	512	485	150	256	tanh	Adam	0.0009	680,450	56.81	0.81	0.30	19e, 19f
4	512	256	1420	150	256	tanh	Adamax	0.001	209,154	56.05	0.82	0.76	19g, 19h
5	512	256	1494	150	400	tanh	Adamax	0.001	209,154	53.67	0.80	0.56	19i, 19j
6	1024	512	811	150	256	tanh	Adamax	0.001	680,450	54.69	0.80	0.39	19k, 19l
7	256	128	1500	150	400	tanh	Adamax	0.001	71,810	52.47	0.79	0.63	19m, 19n

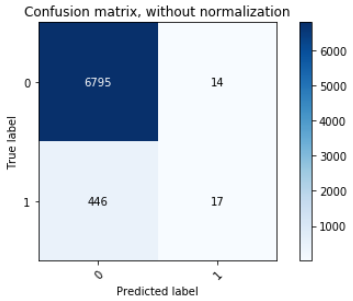
Table 8 shows the results obtained with MLP with three hidden layers using single datafile. Similar hyperparameter cofigurations were tried out as compared to two layered MLP neural network. Experiment 3 in Table 8 lists down hyperparameters that provide the best model performance. These hyperparameter configurations are

similar to the hyperpameter configurations that provided the best result for two layered MLP (experiment 4 in Table 7). The only difference is that three layered MLP model takes less time to train since a bigger batch size is being used. Finally, loss history in Figures 20a, 20c, and 20e for training and validation shows that adding an extra hidden layer on two layered MLP does not lead to overfitting (as long as number of neurons is not increased too much) and provides similar or better results, as shown in Table 8.

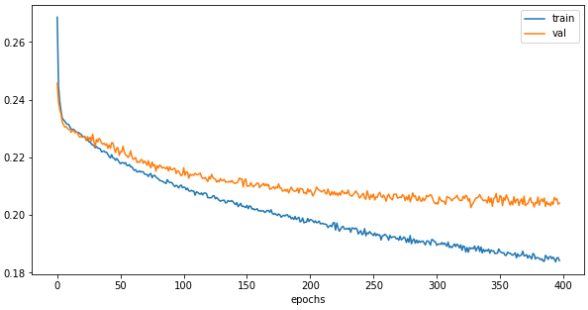
Table 8: MLP with three hidden layers results summary on single datafile.

No.	H_1 units	H_2 units	H_3 units	Epochs	ES	BS	Activation	Optimizer	lr	Model Parameters	Balanced Acc.	AUC	Time	Figures
1	1024	512	256	737	150	300	tanh	Adam	0.001	811256	57.75	0.81	0.41	20a, 20b
2	512	256	128	689	150	256	tanh	Adamax	0.001	241794	57.31	0.81	0.40	20c, 20d
3	512	256	128	877	150	400	tanh	Adamax	0.001	241794	55.78	0.82	0.35	20e, 20d

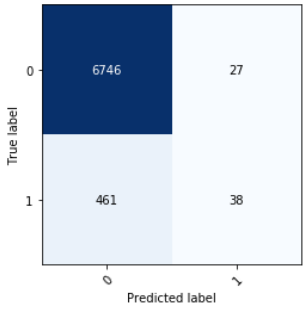
(a) Loss history for experiment 1



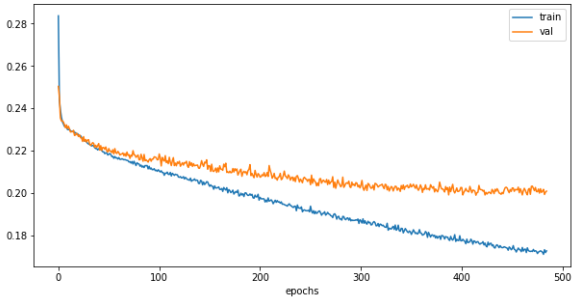
(b) Confusion matrix for experiment 1



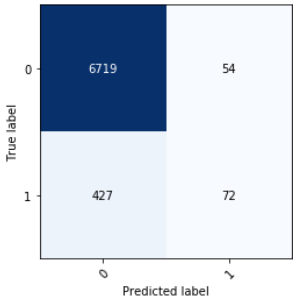
(c) Loss history for experiment 2



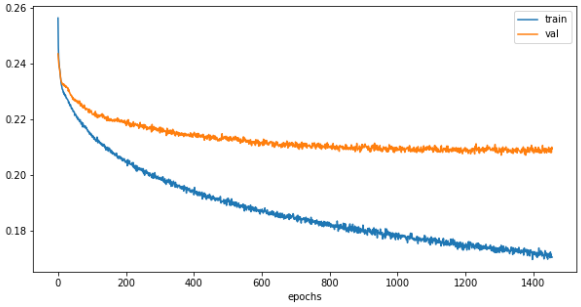
(d) Confusion matrix for experiment 2



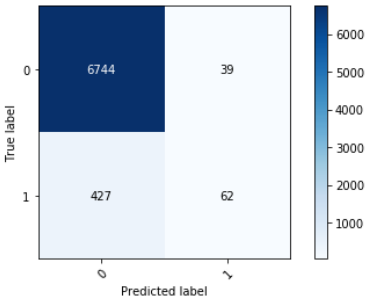
(e) Loss history for experiment 3



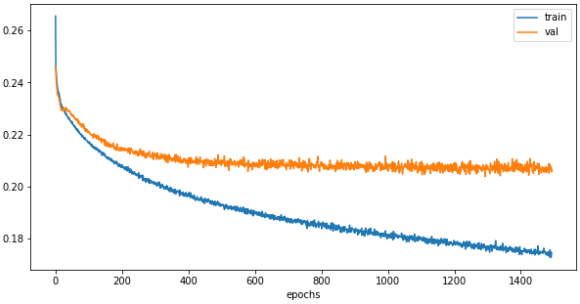
(f) Confusion matrix for experiment 3



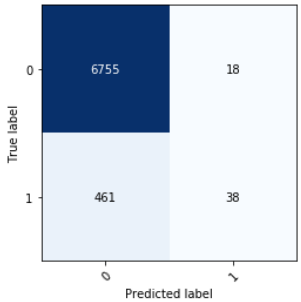
(g) Loss history for experiment 4



(h) Confusion matrix for experiment 4

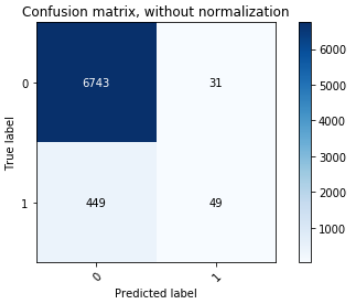


(i) Loss history for experiment 5

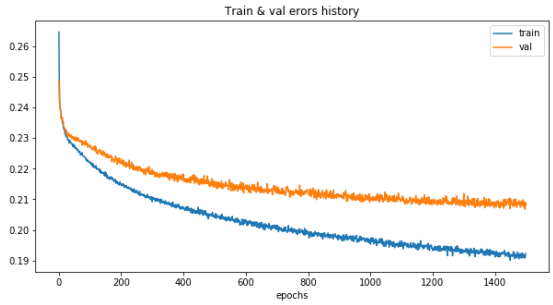


(j) Confusion matrix for experiment 5

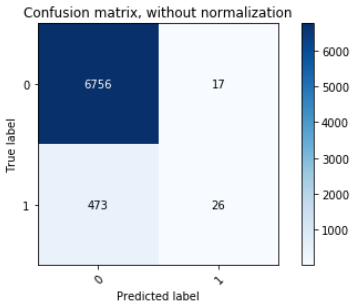
(k) Loss history for experiment 6



(l) Confusion matrix for experiment 6

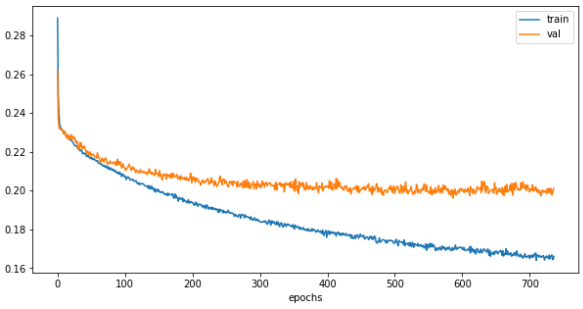


(m) Loss history for experiment 7

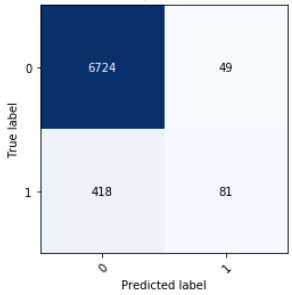


(n) Confusion matrix for experiment 7

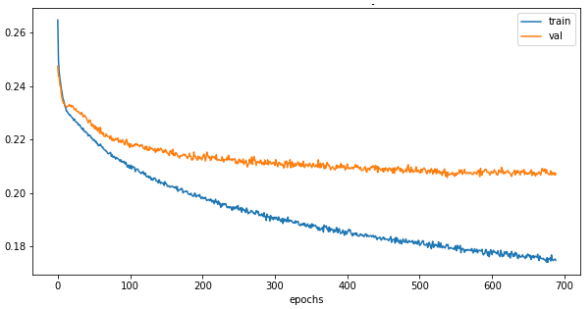
Figure 19: Loss history and confusion matrix for MLP with two hidden layers.



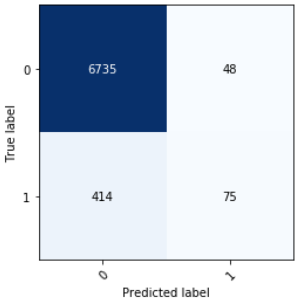
(a) Loss history for experiment 1



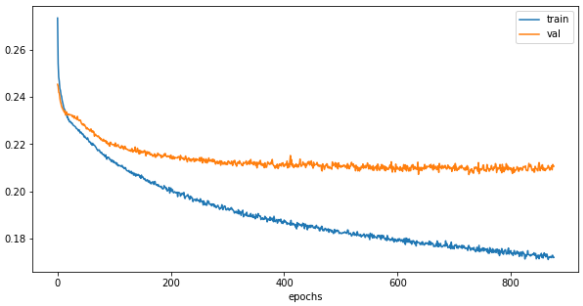
(b) Confusion matrix for experiment 1



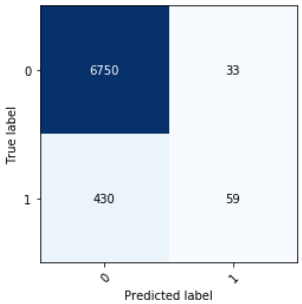
(c) Loss history for experiment 2



(d) Confusion matrix for experiment 2



(e) Loss history for experiment 3



(f) Confusion matrix for experiment 3

Figure 20: Loss history and confusion matrix for MLP with three hidden layers.

4.1.4 XGBoost

After recurrent neural netrworks and artificial neural network, boosted decision trees were trained using single datafile. Table 9 summarizes the results using different hyperparameter values in XGBoost. A short description of hyperparameters in Table 9 is given below:

- **n_estimator**: total number of subtrees (base learners) to grow.
- **max_depth**: maximum depth of each subtree. In other words, this defines the maximum number of features used in each tree.
- **learning rate**: this represents the boosting learning rate which is used to avoid overfitting. This was originally referred to as shrinkage η by Friedman (2002). Shrinkage basically scales newly added weights by a factor η after each step of tree boosting (Chen and Guestrin, 2016).
- **min_child_weight**: minimum sum of instance weight (Hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning.
- **scale_pos_weight**: this can be particularly useful for imbalanced datasets and generally represents the ratio of positive classes to negative classes.
- **gamma**: helps to control regularization in gradient boosting.

In Table 9, experiment 1 uses the default values provided by Scikit-Learn API for XGBoost classification and one can observer that results are very poor using the default values. Increasing the number of trees and depth of each tree helps with better model performance. Using simpler models (first three experiments), one can see from Figures 21a, 21c, and 21e that validation loss follows training loss very closely. But if depth and number of trees are increase too much (experiment 4 of Table 9), there is a chance of overfitting the training data as can be seen from Figure 21g, where validation loss starts increasing while training loss continues to decrease.

In order to optimize hyperparameter values, grid search was used which is one of the popular methods for hyperparameters optimization. Grid search is an exhaustive searching through a manually specified subset of hyperparameter space for a learning algorithm. For implementing grid search for XGBoost, GridSearchCV[7] by scikit-learn was used with scoring method as 'auc' and 'roc_auc'. Table 10 shows the range of values used for grid search and the optimal values found in the range. Experiment 5 in Table 9 shows the results obtained with these optimal values; one can observe the significant increase in the accuracy from the results.

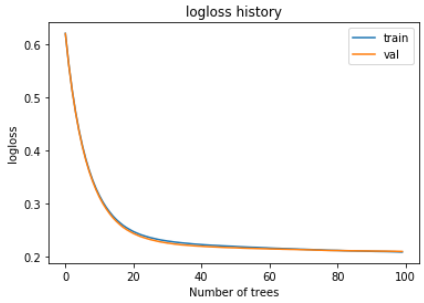
[7] https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html

Table 9: XGBoost results summary using single datafile.

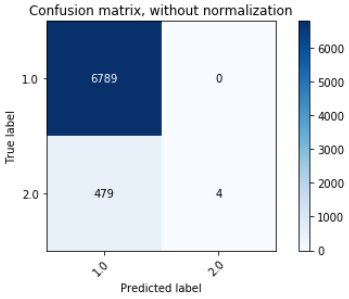
No.	n_estimators	max_depth	min_ child_weight	lr	scale_ pos_weight	gamma	Balanced Acc.	AUC	Time(h)	Figures
1	100	3	1	0.1	1	0	50.28	0.77	0.003	21a, 21b
2	500	3	1	0.1	1	0	52.65	0.81	0.18	21c, 21d
3	500	10	1	0.1	1	0	52.62	0.82	0.20	21e, 21f
4	500	20	1	0.1	1	0	62.36	0.87	0.24	21g, 21h
5	1331	10	1	0.1	1	0	70.92	0.91	0.16	21i, 21j

Table 10: XGBoost hyperparameters optimization using grid search. Using combination of hyperparameters in the specified range, optimal values of each hyperparameter is found as shown in the last column.

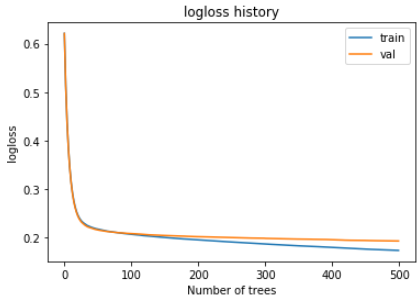
Hyperparameter	**Search range**	**Optimal value**
n_estimaor	[1, 2000]	1331
max_depth	[5, 30]	10
min_child_weight	[1, 6]	1
gamma	[0, 10]	0
scale_pos_weight	[7.5, 14]	9.6



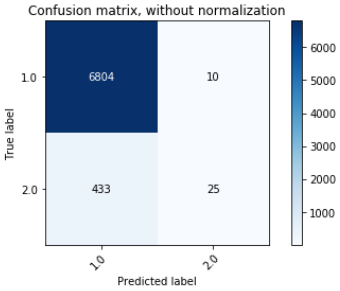
(a) Loss history for experiment 1



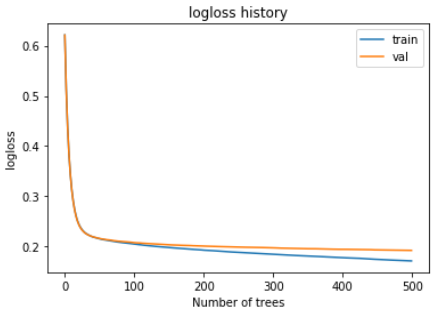
(b) Confusion matrix for experiment 1



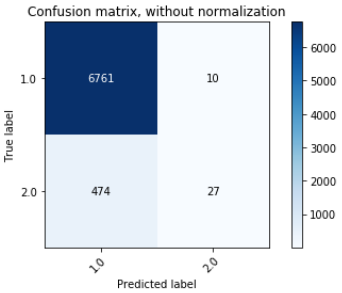
(c) Loss history for experiment 2



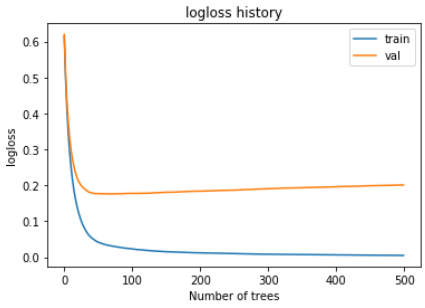
(d) Confusion matrix for experiment 2



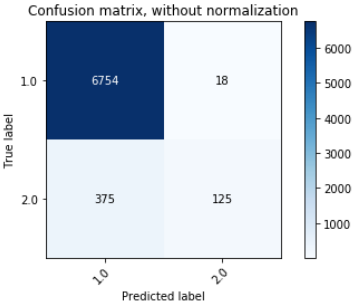
(e) Loss history for experiment 3



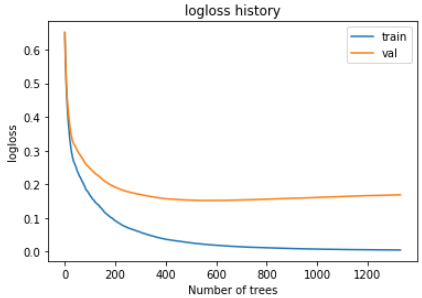
(f) Confusion matrix for experiment 3



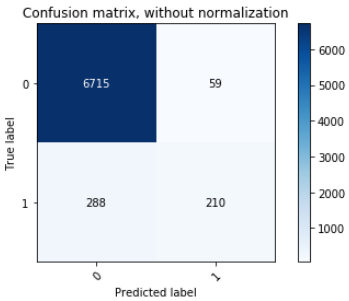
(g) Loss history for experiment 4



(h) Confusion matrix for experiment 4



(i) Loss history for experiment 5



(j) Confusion matrix for experiment 5

Figure 21: Loss history and confusion matrix for XGBoost.

4.2 Using Combined Dataset

After experimenting with one datafile, the next step was to use data from multiple files and evaluate how models perform on bigger dataset. For this purpose, total of 10 CSV files were chosen. Table 11 shows a comprehensive summary of the sources files used to merge the data. One can observe from Table 11 that weight (%) of *ACK* and *NACK* remains almost the same through all the files. For training, validation, and testing, data split ratio was 85%, 9%, and 6% respectively as shown in Table 12.

Table 11: Summary of combined dataset. First 10 rows represent the single datafiles that have been processed individually and then merged to create a bigger dataset as shown in the last row of the table.

Index	Number of Samples	Total ACKs	Total NACKs	ACK %	NACK %	Total users
1	90902	84797	6105	93.28	6.72	1121
2	99416	92637	6779	93.18	6.82	1104
3	107017	100639	6378	94.04	5.96	959
4	97143	91288	5855	93.97	6.03	1129
5	92629	87279	5350	94.22	5.78	1067
6	100665	92487	8178	91.88	8.12	755
7	88238	83344	4894	94.45	5.55	963
8	88363	82298	6065	93.14	6.86	725
9	100068	93121	6947	93.06	6.94	769
10	89948	83633	6315	92.98	7.02	544
Combined	954389	891494	62894	93.41	6.59	9136

Table 12: Data split ratio for training, validation and testing. 85% data is used for training while the rest 15% is used for validation and testing.

Dataset	**Split ratio(%)**
Training data	85.00
Validation data	9.00
Testing data	6.00

The following subsections show the results of using LSTM, GRU, MLP, and XGBoost on combined dataset.

4.2.1 Long Short-Term Memory (LSTM)

The section enlists the results of different hyperparameters configuration using LSTM on combined dataset. Table 13 indicates the hyperparameter values and Figure 22 shows the corresponding figures for loss history and confusion matrix. For detailed description of hyperparameters used in Table 13, one can refer to Section 4.1.1.

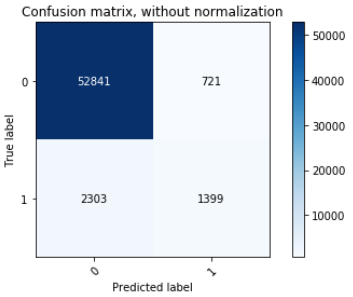
From the results in Table 13, one can notice that experiment 1 gives an AUC score of 0.86 which is quite good. But if one looks at loss history for this experiment in Figure 22a, after certain number of epochs the gap between training and validation

loss increases. This is not an indicative of model overfitting since early stopping is used to stop such a behavior, but it does indicate that underlying model is a bit complex; therefore reduced complexity might help to over come such a gap between training and validation loss. One can observe from hyperparameters configuration in experiment 2 till experiment 7 that number of LSTM units and LSTM layers were reduced to decrease the model complexity. Loss history plots in Figures 22c, 22e, 22g, 22i, 22k, and 22m show that validation loss curve starts following training loss more closely as compared to Figure 22a. But with reduced complexity, model performance also decreases (as shown by AUC scores). Finally, this problem was solved using *class weights* parameter in LSTM model. Experiment number 8 and 9 show that *class weights* parameter can be really useful to overcome the gap with no impact on model accuracy. One can notice from Figures 22o and 22q that validation loss decreases as training loss decreases and also the gap between two loss curves has decreased significantly as compared to Figures 22a and 22c where except *class weights*, same hyperparameters were used.

Table 13: LSTM experiments on combined dataset.

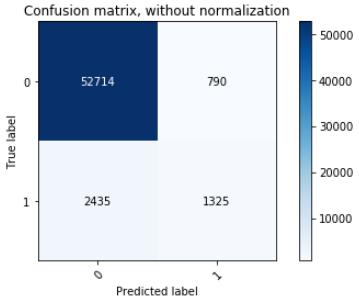
No.	LSTM Units	LSTM Layers	Epochs	ES	BS	CW	Recurrent Dropout	Optimizer	lr	Model Parameters	Balanced Acc	AUC	Time	Figures
1	256	3	206	100	8192	False	0.30	Adam	0.001	1,324,546	68.22	0.86	1.34	22a, 22b
2	128	3	330	100	8192	False	0.30	Adam	0.001	334,594	66.88	0.85	1.204	22c, 22d
3	64	3	490	100	8192	False	0.30	Adam	0.001	85,378	61.34	0.84	1.59	22e, 22f
4	64	3	512	100	16384	False	0.30	Adam	0.001	85,378	62.20	0.84	1.729	22g, 22h
5	64	3	482	100	8192	False	0.50	Adam	0.001	85,378	61.47	0.84	1.62	22i, 22j
6	64	2	940	100	8192	False	0.30	Adam	0.001	52,354	60.18	0.84	2.25	22k, 22l
7	64	2	1235	100	16384	False	0.30	Adam	0.001	52,354	59.81	0.83	3.65	22m, 22n
8	256	3	321	100	8192	True	0.30	Adam	0.001	1,324,546	76.69	0.86	2.09	22o, 22p
9	128	3	455	100	8192	True	0.30	Adam	0.001	334,594	75.63	0.85	1.78	22q, 22r

(a) Loss history for experiment 1

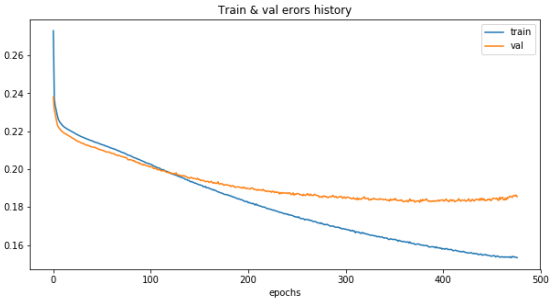


(b) Confusion matrix for experiment 1

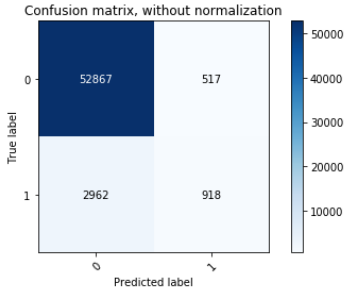
(c) Loss history for experiment 2



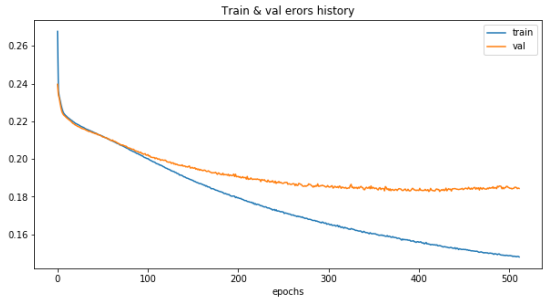
(d) Confusion matrix for experiment 2



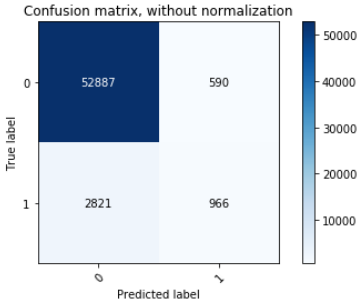
(e) Loss history for experiment 3



(f) Confusion matrix for experiment 3

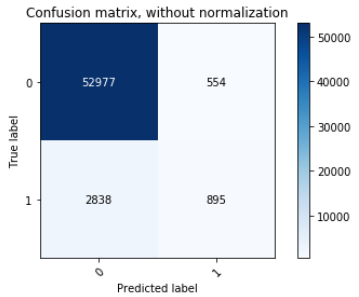


(g) Loss history for experiment 4



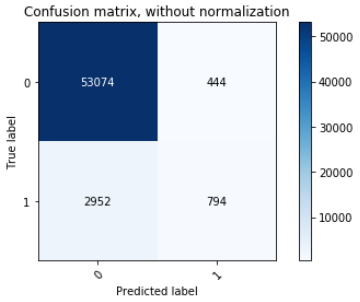
(h) Confusion matrix for experiment 4

(i) Loss history for experiment 5

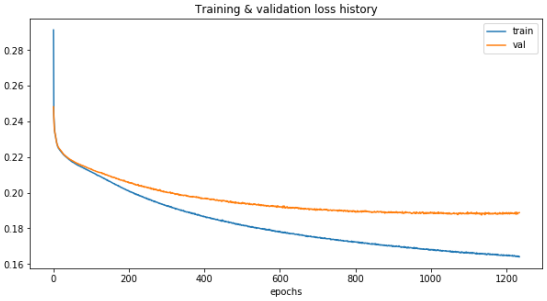


(j) Confusion matrix for experiment 5

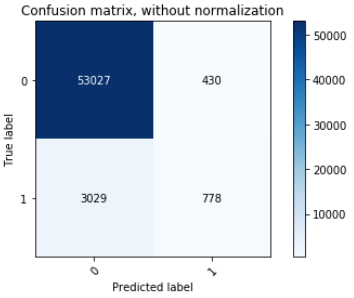
(k) Loss history for experiment 6



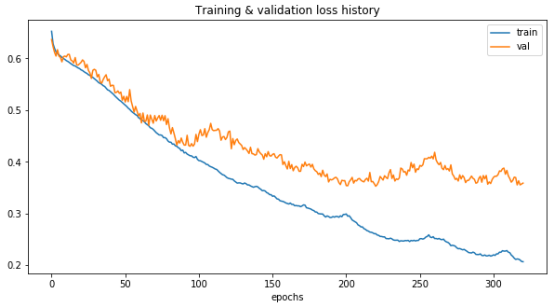
(l) Confusion matrix for experiment 6



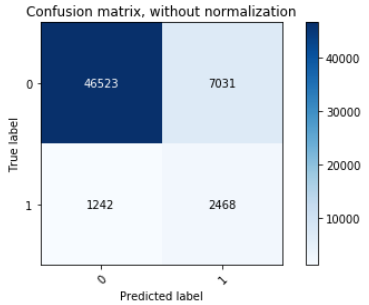
(m) Loss history for experiment 7



(n) Confusion matrix for experiment 7

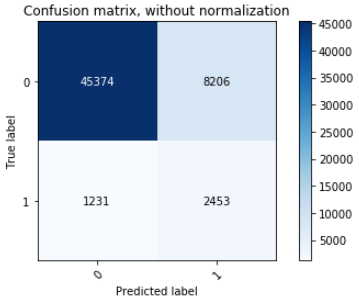


(o) Loss history for experiment 8



(p) Confusion matrix for experiment 8

(q) Loss history for experiment 9



(r) Confusion matrix for experiment 9

Figure 22: Loss history and confusion matrix for LSTM on combined dataset.

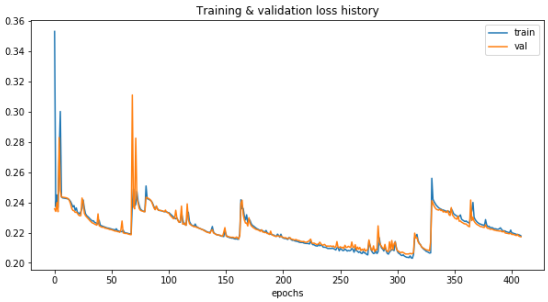
4.2.2 Gated Recurrent Units (GRUs)

As compared to results on single dataset, GRU model performance on combined dataset was quite poor. Different combination of hyperparameter values were tried out, from different layers to number of GRU units, batch size and so on. But still GRU performance remained poor as can be seen from AUC score in Table 14. Similar to LSTMs, using *class weights* helped with better model performance in GRU as well.

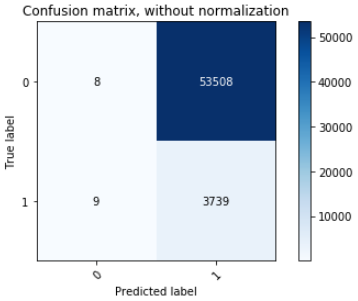
One can observe from Figures 23a, 23c, 23g, and 23i that training and validation loss do not decrease continuously rather abrupt behavior is observed with decrease in loss followed by sudden increase and the pattern continues. In Figure 23e though, sudden decrease in loss is observed at the start of training and then the model stops learning for the rest of epochs.

Table 14: GRU results summary on combined dataset.

No.	GRU Units	GRU Layers	Epochs	ES	BS	CW	Recurrent dropouts	Optimizer	lr	Model Parameters	Balanced Acc.	AUC	Time(h)	Figures
1	128	3	450	100	8192	False	0.30	Adam	0.001	251,010	49.00	0.26	1.36	23a, 23b
2	128	3	117	100	8192	True	0.30	Adam	0.001	251,010	69.17	0.75	0.37	23c, 23d
3	256	3	188	100	8192	True	0.30	Adam	0.001	993,538	32.18	0.26	1.48	23e, 23f
4	256	3	233	100	20000	True	0.30	Adam	0.001	993,538	66.49	0.73	1.42	23g, 23h
5	128	2	638	100	8192	True	0.30	Adam	0.001	152,322	63.58	0.67	1.39	23i, 23j

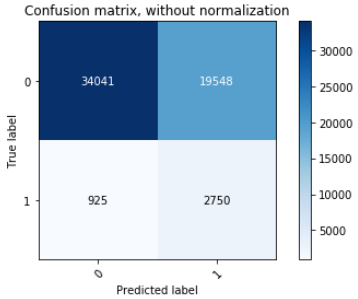


(a) Loss history for experiment 1

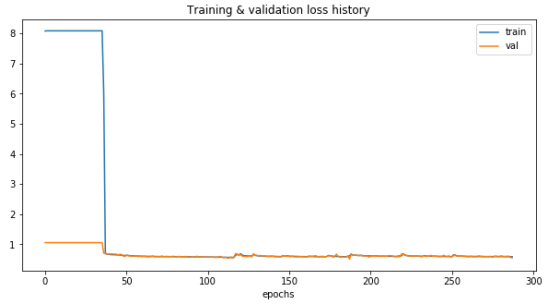


(b) Confusion matrix for experiment 1

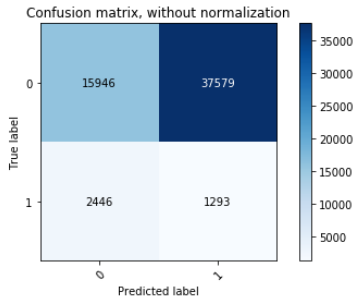
(c) Loss history for experiment 2



(d) Confusion matrix for experiment 1

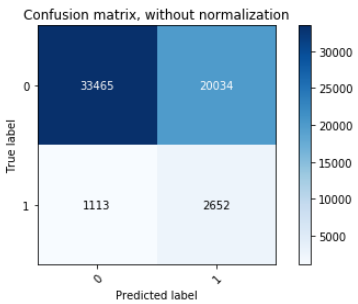


(e) Loss history for experiment 3



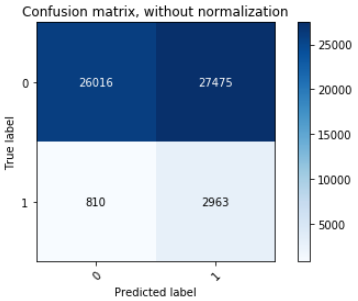
(f) Confusion matrix for experiment 3

(g) Loss history for experiment 4



(h) Confusion matrix for experiment 4

(i) Loss history for experiment 5



(j) Confusion matrix for experiment 5

Figure 23: Loss history and confusion matrix for GRU on combined dataset.

4.2.3 Multilayer Perceptron (MLP)

After experimenting with LSTM and GRU, the next step was to implement MLP on combined dataset. Table 15, 16, and 17 show the results of trained MLP model using two, three and four hidden layers respectively.

One can see from experiments 1 till 6 in Table 16 that different batch sizes were used to determine the batch size that gives the best performance. Among different activation functions that were used, *tanh* and ELU (exponential linear units) (Clevert et al., 2015) provided better results, although ELU activation function was slightly better than *tanh*. Moreover, among Adam, Adamax, SGD, and RMSProp optimizers, Adam and Adamax were better choices in terms of accuracy. Finally, one can see from experiment 12 and 13 in Table 16 that decreasing number of neurons in hidden layers reduces model accuracy slightly while too much increase in number of neurons impedes model training as is evident from Figures 25y and 25z.

In order to properly tune the model, MLP model with three hidden layers was explored in quite detail with different values of batch sizes, activation and number of units. From the experiments, it seems that an MLP model with three hidden layers is sufficient to learn the model complexity. Using two layered MLP model, model performance decreases slightly as shown in Table 15. With four layered MLP, results similar to three layered MLP were obtained. With increased number of hidden layers, model usually overfits the training data and does not generalize well (Goodfellow et al., 2016). Therefore, it is always preferable to settle for a less complex model as compared to a complex model when there is no increase in accuracy.

One important thing to observe from loss history of two, three and four layered MLP models is that validation loss curve follows very closely the training loss curve. This can be seen from all loss curves in Figures 24, 25, and 26 (except Figure 25y where model never learned). This is quite noticeable as compared to loss history obtained from LSTMs, GRUs, and XGBoost models where after certain epochs, gap between training and validation loss starts to increase.

Table 15: MLP with two hidden layers on combined dataset.

No.	H_1 units	H_2 units	Epochs	BS	Activation	Optimizer	lr	Model Parameters	Balanced Acc.	AUC	Time	Figures
1	1024	256	2178	20000	elu	Adam	0.001	680450	53.55	0.80	1.62	24a, 24b

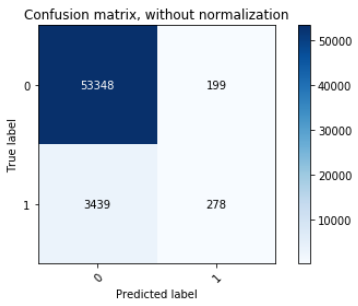
Table 16: MLP with three hidden layers on combined dataset.

No.	H_1 units	H_2 units	H_3 units	Epochs	BS	Activation	Optimizer	lr	Model Parameters	Balanced Acc.	AUC	Time	Fig	CM
1	1024	512	256	910	600	tanh	Adam	0.001	811266	50.12	0.75	2.86	25a	25b
2	1024	512	256	564	1000	tanh	Adam	0.001	811266	50.45	0.77	1.17	25c	25d
3	1024	512	256	636	10000	tanh	Adam	0.001	811266	52.76	0.80	0.55	25e	25f
4	1024	512	256	1003	20000	tanh	Adam	0.001	811266	52.75	0.81	0.77	25g	25h
5	1024	512	256	825	40000	tanh	Adam	0.001	811266	52.38	0.80	0.88	25i	25j
6	1024	512	256	930	50000	tanh	Adam	0.001	811266	52.30	0.80	0.81	25k	25l
7	1024	512	256	573	20000	relu	Adam	0.001	811266	53.15	0.79	0.47	25m	25n
8	1024	512	256	1899	20000	elu	Adam	0.001	811266	56.35	0.83	1.56	25o	25p
9	1024	512	256	2092	20000	elu	Adamax	0.001	811266	54.03	0.82	1.72	25q	25r
10	1024	512	256	3000	20000	elu	SGD	0.001	811266	50.00	0.67	2.43	25s	25t
11	1024	512	256	2101	20000	elu	RMSprop	0.001	811266	55.16	0.82	1.72	25u	25v
12	512	256	128	1293	20000	elu	Adam	0.001	241794	51.93	0.80	0.38	25w	25x
13	2048	1024	512	101	20000	elu	Adam	0.001	2933250	50.00	0.50	0.068	25y	25z

Table 17: MLP with four hidden layers on combined dataset.

No.	H_1 units	H_2 units	H_3 units	H_4 units	Epochs	BS	Activation	Optimizer	lr	Model Parameters	Balanced Acc.	AUC	Time	Figures
1	256	128	64	32	2495	20000	elu	Adam	0.001	81954	51.31	0.79	0.64	26a, 26b
2	1024	512	256	128	2168	20000	elu	Adam	0.001	843906	57.42	0.83	1.83	26c, 26d

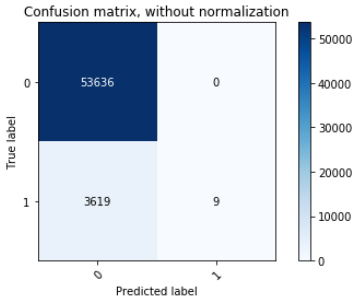
(a) Loss history for experiment 1



(b) Confusion matrix for experiment 1

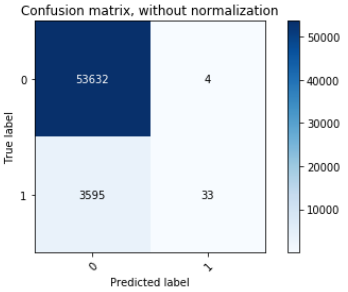
Figure 24: Loss history and confusion matrix for MLP with 2 hidden layers.

(a) Loss history for experiment 1

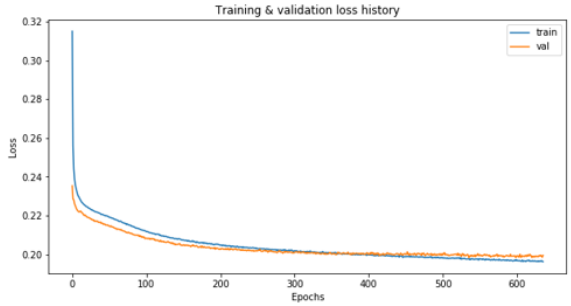


(b) Confusion matrix for experiment 1

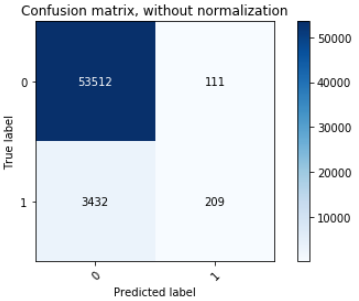
(c) Loss history for experiment 2



(d) Confusion matrix for experiment 2

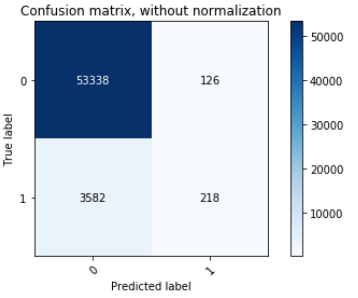


(e) Loss history for experiment 3

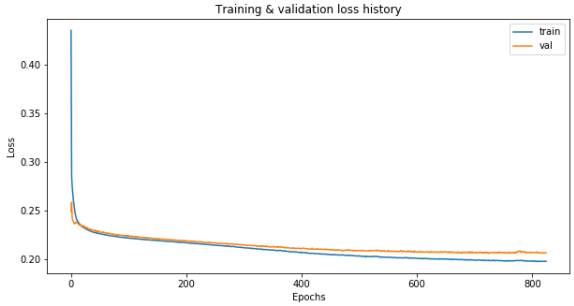


(f) Confusion matrix for experiment 3

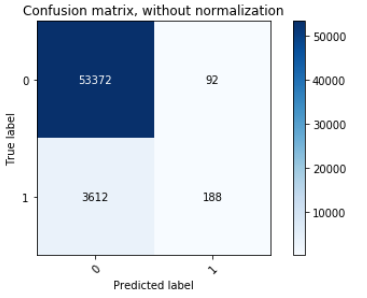
(g) Loss history for experiment 4



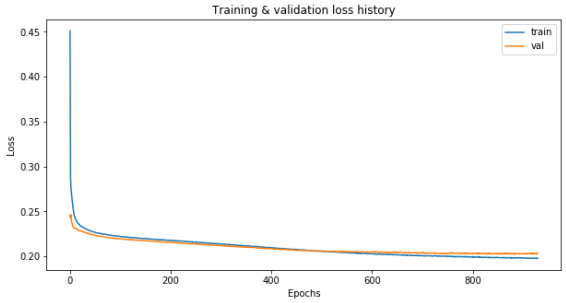
(h) Confusion matrix for experiment 4



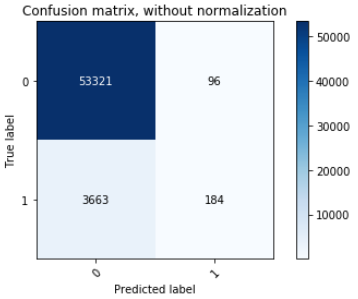
(i) Loss history for experiment 5



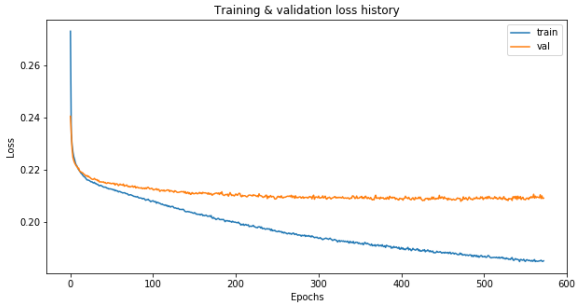
(j) Confusion matrix for experiment 5



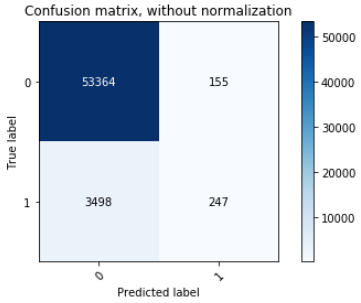
(k) Loss history for experiment 6



(l) Confusion matrix for experiment 6

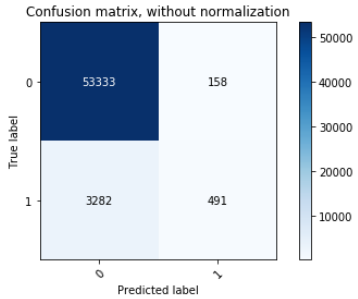


(m) Loss history for experiment 7



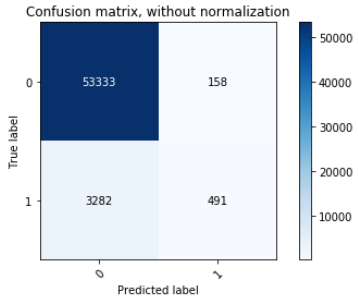
(n) Confusion matrix for experiment 7

(o) Loss history for experiment 8

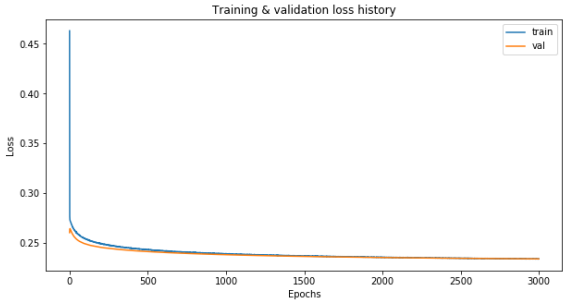


(p) Confusion matrix for experiment 8

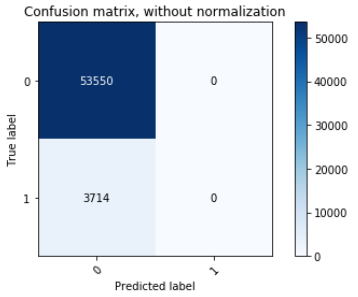
(q) Loss history for experiment 9



(r) Confusion matrix for experiment 9

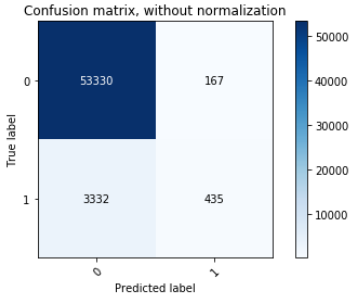


(s) Loss history for experiment 10

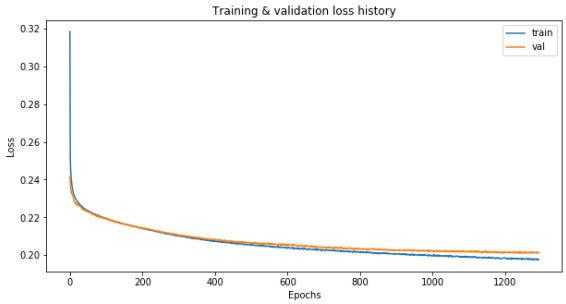


(t) Confusion matrix for experiment 10

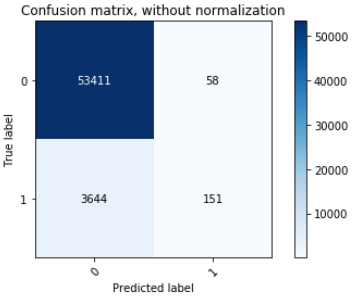
(u) Loss history for experiment 11



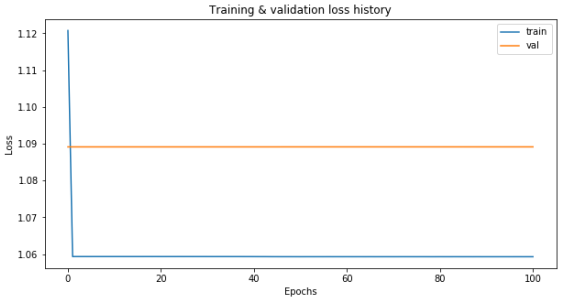
(v) Confusion matrix for experiment 11



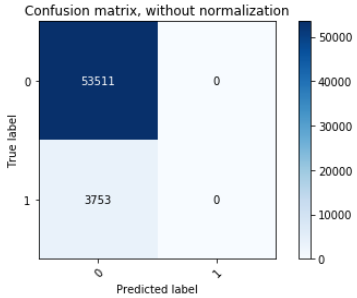
(w) Loss history for experiment 12



(x) Confusion matrix for experiment 12

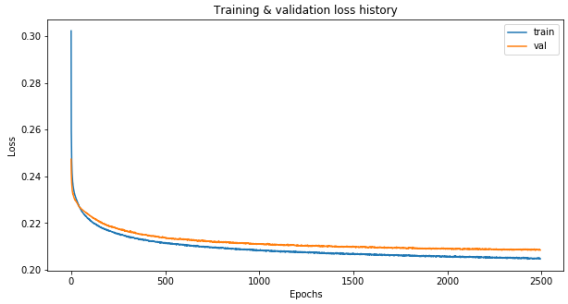


(y) Loss history for experiment 13

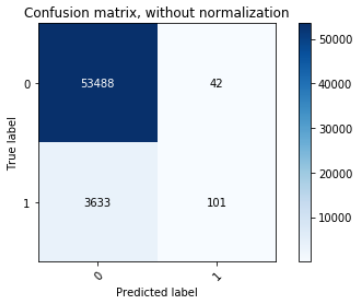


(z) Confusion matrix for experiment 13

Figure 25: Loss history and confusion matrix for MLP with 3 hidden layers.

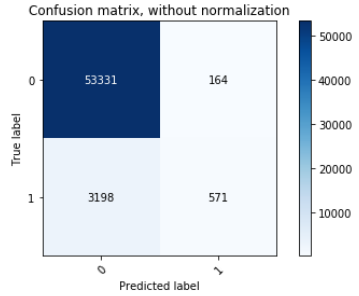


(a) Loss history for experiment 1



(b) Confusion matrix for experiment 1

(c) Loss history for experiment 2



(d) Confusion matrix for experiment 2

Figure 26: Loss history and confusion matrix for MLP with 4 hidden layers.

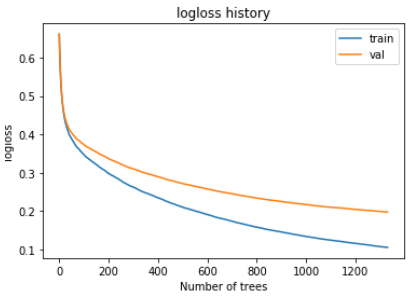
4.2.4 XGBoost

Finally, in this section, one can find the results obtained with XGBoost on combined dataset summarized in Table 18. Similar to Section 4.1.4, different hyperparameter values were experimented with. One can refer to the same Section 4.1.4 for detailed description of each hyperparameter in Table 18.

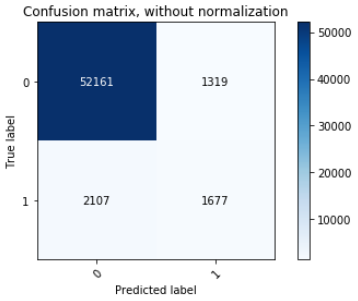
Experiment 1 in Table 18 uses the same hyperparameters that gave the best results using single datafile. Using these configurations, quite good model performance was achieved. In order to tune the XGBoost model even better, parameters related to number of trees and maximum depth of each tree were experimented with. From the experiments, it seems that one needs to find a balance between these two hyperparameters. This can be seen from the results of experiment 2 and experiment 5 where same model performance is achieved. In experiment 2, fewer decision trees with more depth are used whereas in experiment 5, more decision trees with less depth are used. Generally, it is a good idea not to increase both number of trees and tree depth too much since it will results in a very complex model. Figures 27a, 27c, 27e, 27g, and 27i correspond to loss history for each of the experiment in Table 18; whereas Figures 27b, 27d, 27f, 27h, and 27j correspond to confusion matrix for the binary XGBoost classifier for each of the experiment in Table 18.

Table 18: XGBoost experiments on combined dataset.

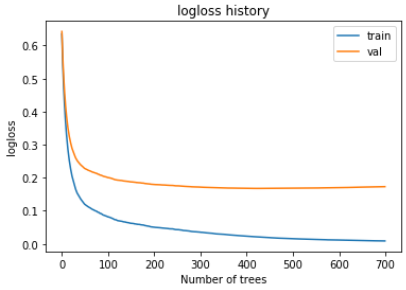
No.	n_estimators	max_depth	min_ child_weight	lr	scale_ pos_weight	gamma	Balanced Acc.	AUC	Time(h)	Figures
1	1331	10	1	0.1	9.6	0	70.92	0.87	1.93	27a, 27b
2	700	20	1	0.1	9.6	0	63.97	0.88	2.37	27c, 27d
3	500	20	1	0.05	9.6	0	64.14	0.86	2.38	27e, 27f
4	1000	10	1	0.05	9.6	0	72.37	0.86	1.81	27g, 27h
5	1000	15	1	0.1	9.6	0	66.81	0.88	2.99	27i, 27j



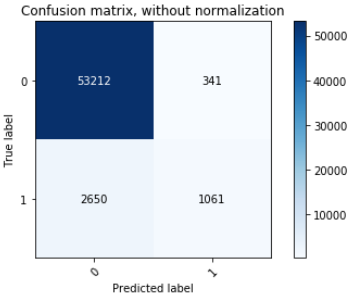
(a) Loss history for experiment 1



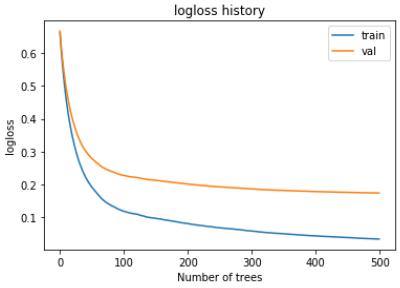
(b) Confusion matrix for experiment 1



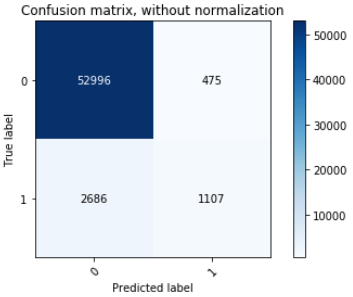
(c) Loss history for experiment 2



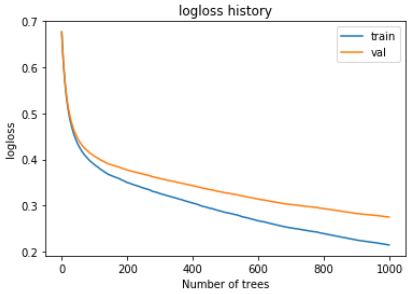
(d) Confusion matrix for experiment 2



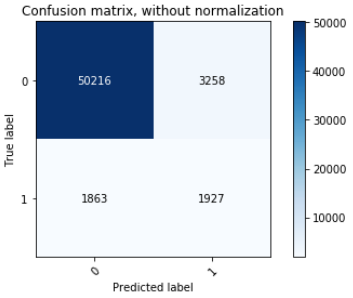
(e) Loss history for experiment 3



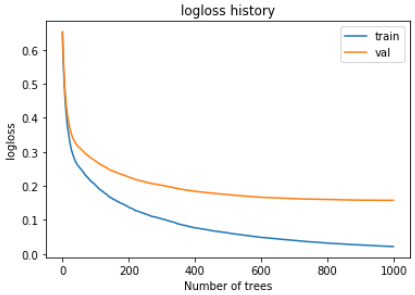
(f) Confusion matrix for experiment 3



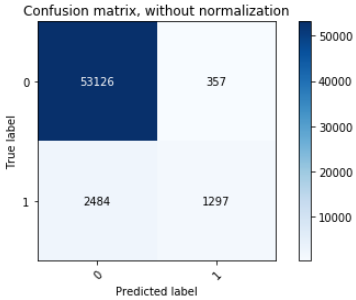
(g) Loss history for experiment 4



(h) Confusion matrix for experiment 4



(i) Loss history for experiment 5



(j) Confusion matrix for experiment 5

Figure 27: Loss history and confusion matrix for XGBoost.

4.3 Feature Selection

Feature selection (Guyon and Elisseeff, 2003) is a method in machine learning to reduce redundant data and noise by selecting features that contribute the most to model learning and discarding irrelevant features that might have either zero or negative impact on model learning. The motivation is to reduce model complexity by using relevant features only, thereby avoiding the risk of overfitting and reducing training time. For the research problem at hand, as mentioned in Section 3.1, originally the dataset contained around 314 features which were narrowed down to 10 features by the domain experts. The motivation behind feature selection in this scenario is to identify possible features that domain experts think as useful but machine learning identifies them as redundant or bad features. Although there are numerous ways to perform and implement feature selection in Python, we have primarily used the following two methods:

1. By using *feature_importances_* property of the trained XGBoost model.
2. By dropping individual features, or combination of features to evaluate the impact on model performance. For this part, we only use XGBoost and MLP models since LSTM/GRU take significantly longer time for training.

*feature_importances_* property in XGBoost model lists down the features that it found useful (during learning phase) for classification/regression depending upon the *importance_type*. *Importance_type* is a method to filter the features according to certain criterias, namely as gain, coverage, and frequency. Details of these criteria follow as below:

1. **Gain**: This represents the relative importance of corresponding feature to the model calculated by taking each feature's contribution for each tree. The higher the gain number, the more important the feature is for generating the prediction.
2. **Coverage**: This represents the relative number of observations related to the corresponding feature. For example, let us say that we had 100 observations with K features and we constructed a model with 3 trees. Now if a feature f_i was used to decide the leaf node for m different observations in all trees combined, then coverage will be calculated as m expressed as percentage of sum of all K features' coverages.
3. **Frequency/Weight**: This represents the relative number of times a particular feature occurs in the trees of the model.

Since we are interested in finding features that help the model with prediction, we will use only *gain* as importance type.

Experiments followed in this section for XGBoost use the same hyperparameter values as described in experiment no. 5 in Table 9. However we will use different past window sizes to observe if varying window sizes affect the feature importance.

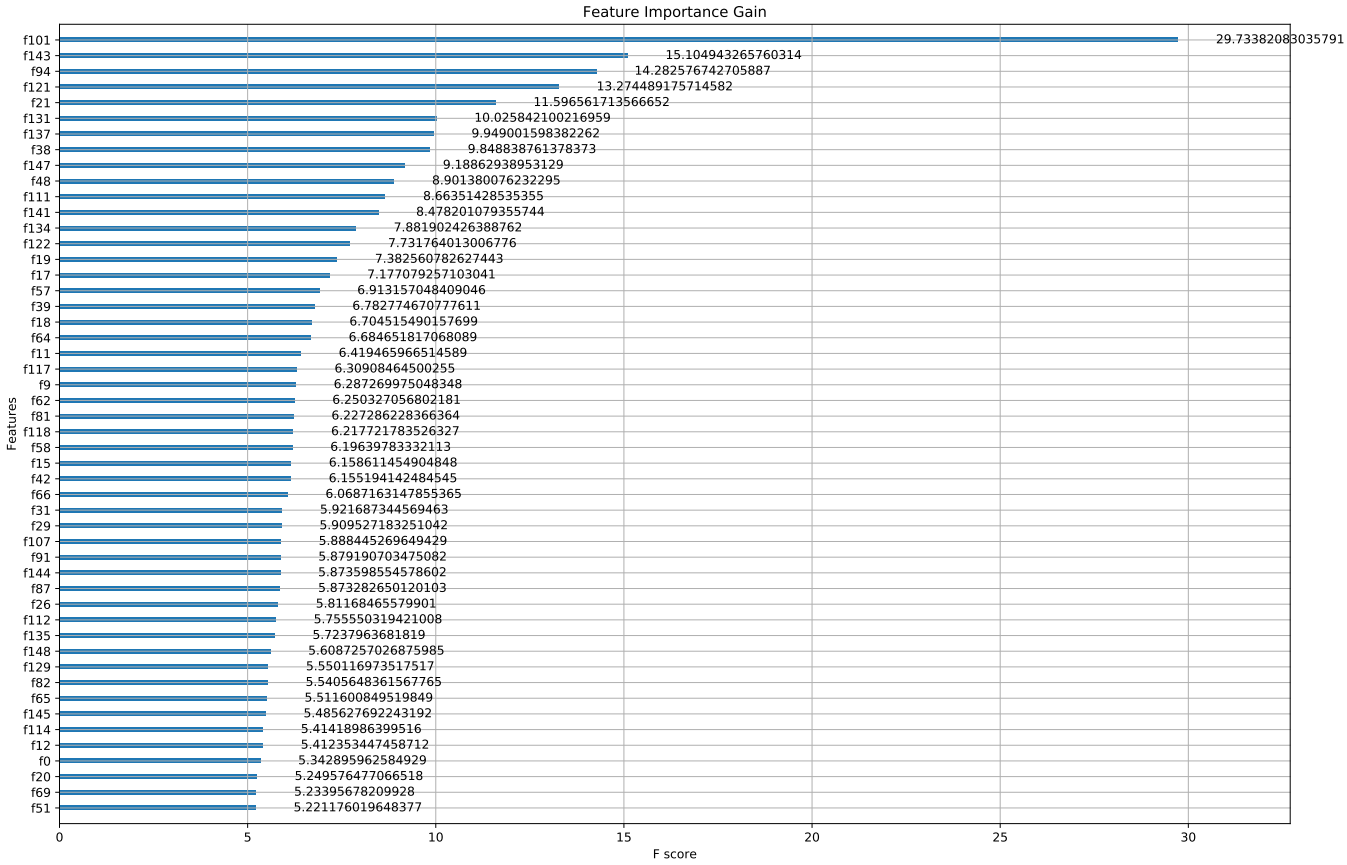


Figure 28: Feature importance plot

Figure 28 shows the results of feature importance plot generated by XGBoost inbuilt function, using window size of 15 and selecting the top 50 features only. The plot is not really useful in visualizing the feature importance and gets cluttered if more top n features are visualized. In order to overcome this issue, a solution was designed to score each feature according to the scores obtained using *feature_importances_* property in trained XGBoost model. In order to explain this better, let us assume that we are using a window size of 15 and have total of 10 features, then we have 150 indices with different scores. Each feature contributes 15 different values over the past (hence window size = 15). In order to get contribution from a feature f_i, one needs to iterate over 150 indices and find the indices that belong to feature f_i and sum their scores. Now, it is also important how one sums these scores. Since in our research problem, it is desirable to use as less values from the past as possible, a penalty parameter is introduced which penalizes the past values more as compared to recent values. The idea is to weigh recent contributions (in terms of time lapse)

from features more than contributions that happened long time ago. Equation (23) shows the mathematical expression for the function used to penalize values from the past while Figure 29 shows the function plot for window size of 15.

$$f(x) = \frac{1}{\exp(\frac{x}{s^{1.5}})} \tag{23}$$

where s denotes the window size, that is total number of past values used and x denotes the index of features listed in descending order with respect to importance score.

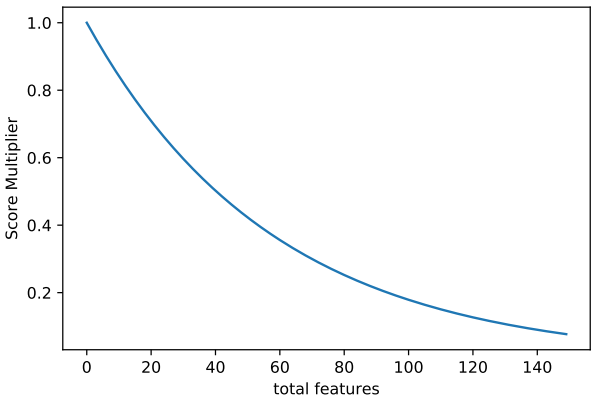


Figure 29: Line plot for penalty function with window size of 15.

Using the penalty function and score of features (as shown in Figure 28), one can generate a bar plot for all features with their respective scores. Figure 30, 31, 32 ,33, 34, and 35 show the results for window size of 5, 10, 15, 20, 25, and 50 respectively.

It can be observed from the results that *ETtiTraceDlParUe_rrmDeltaCqiCw0* is the most important radio parameter regardless of the window size being used. On the other hand, *ETtiTraceDlParUe_wbCqiCompensateCw0* and *ETtiTraceDlParUe_averCirRgbCw0* seem to be least important. Moreover, it can be seen that for some features, their importance vary with varying window sizes.

In order to verify the findings and to explore further the importance of features, experiments were conducted with MLP and XGBoost where individual features or subset of features were dropped to observe their relative contribution towards the model performance. Table 19, 20, and 21 show a brief summary where experiments were conducted 3 times and results were averaged.

Table 19 shows the results of dropping individual features and their impact on the model performance. The most significant drop in performance, for both XGBoost and MLP, is observed when *ETtiTraceDlParUe_wbCqiCompensateCw0* feature is dropped. This is also corroborated by the *feature_importances_* analysis using XGBoost that *ETtiTraceDlParUe_wbCqiCompensateCw0* seems to be the most relevant feature. On the other hand, *ETtiTraceDlParUe_mcsIndexCw1*, *ETtiTraceDlParUe_ModulationCw1* and *EHarqParDl_rrmRecommendedMcsCw1* seem to be have the least effect on model learning when dropped.

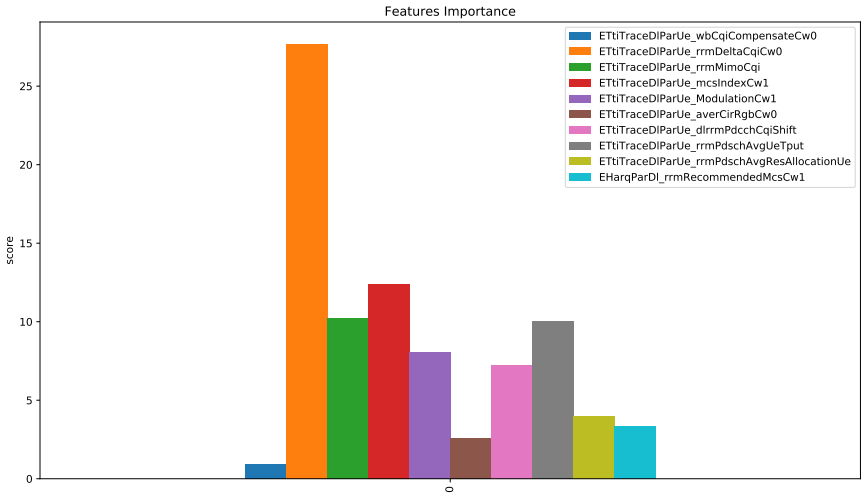


Figure 30: Feature importance plot with window size 5.

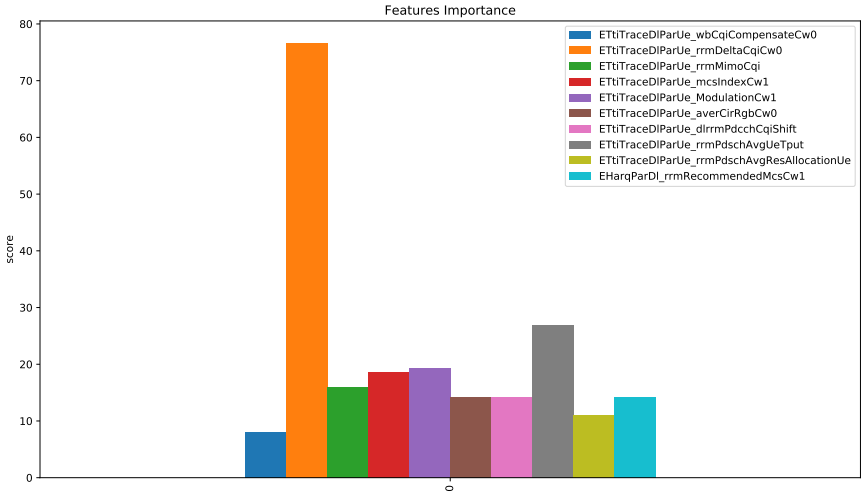


Figure 31: Feature importance plot with window size 10.

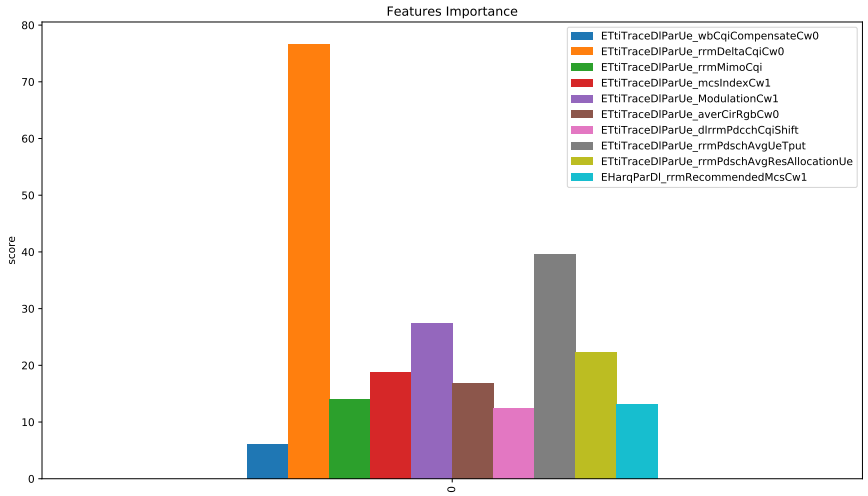


Figure 32: Feature importance plot with window size 15.

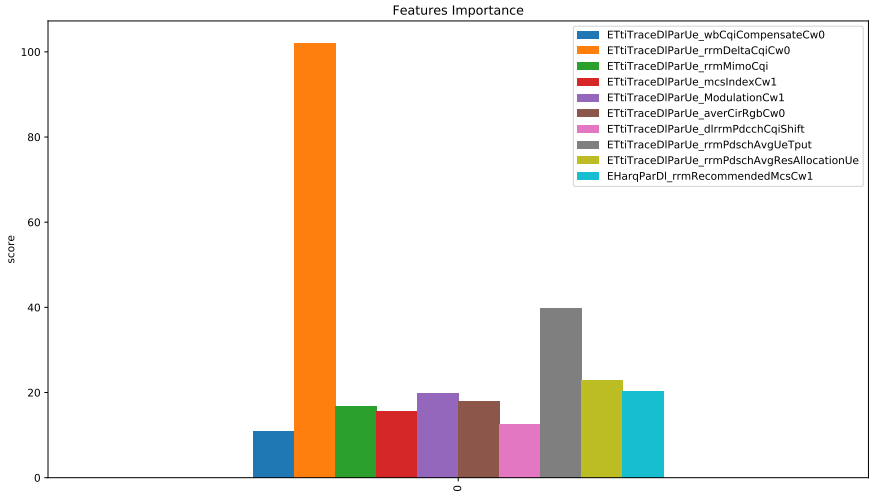


Figure 33: Feature importance plot with window size 20.

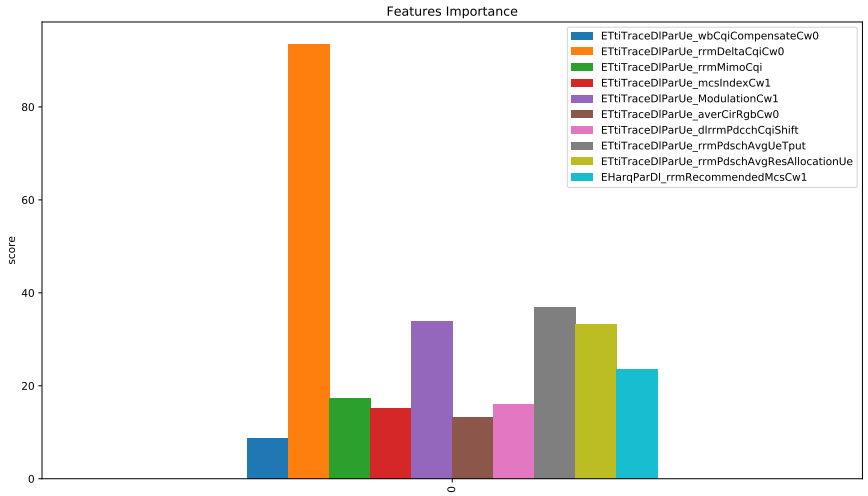


Figure 34: Feature importance plot with window size 25.

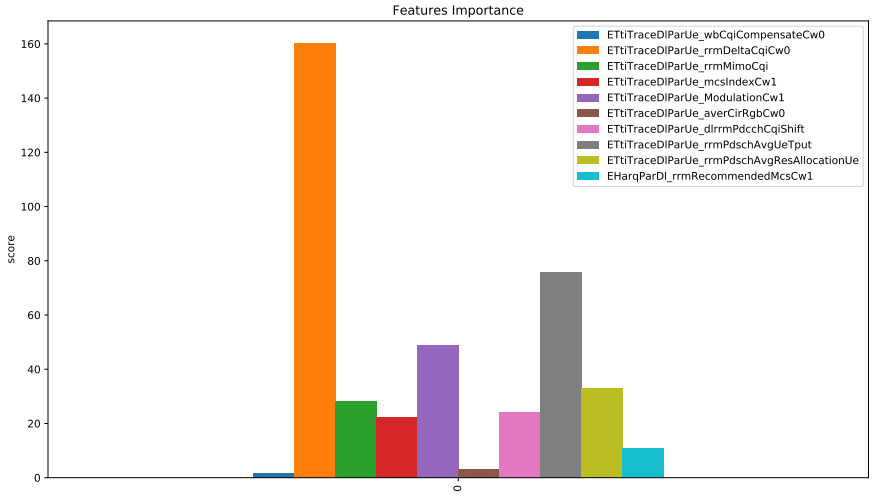


Figure 35: Feature importance plot with window size 50.

Table 19: Experiments with dropped features. Each individual feature is dropped and its impact on model performance is evaluated.

Index.	Dropped features	XGBoost		MLP	
		Balanced Accuracy (%)	AUC	Balanced Accuracy (%)	AUC
Reference		**70**	**0.91**	**57.42**	**0.82**
1	ETtiTraceDlParUe_wbCqiCompensateCw0	69.95	0.90	55	0.81
2	ETtiTraceDlParUe_rrmDeltaCqiCw0	66.88	0.87	53.24	0.75
3	ETtiTraceDlParUe_rrmMimoCqi	68.34	0.88	54.72	0.81
4	ETtiTraceDlParUe_mcsIndexCw1	69.63	0.91	57.03	0.81
5	ETtiTraceDlParUe_ModulationCw1	69.89	0.91	56.15	0.81
6	ETtiTraceDlParUe_averCirRgbCw0	70.61	0.89	56.90	0.82
7	ETtiTraceDlParUe_dlrrmPdcchCqiShift	69.63	0.89	54.37	0.79
8	ETtiTraceDlParUe_rrmPdschAvgUeTput	68.75	0.89	55.70	0.81
9	ETtiTraceDlParUe_rrmPdschAvgResAllocationUe	70.15	0.90	57.80	0.80
10	EHarqParDl_rrmRecommendedMcsCw1	68.89	0.90	55.56	0.81

The next step was to try different combinations of the weak contributors and evaluate their performance, as shown in Table 20. Any feature that decreases the model AUC score, when dropped, by no more than 0.01 can be considered as a weak feature. Experiments 11, 12, 13 and 15 in Table 20 show that such combination of features (under column 1) could be dropped with a minimum impact on model performance. The rest of the experiments in Table 20 (in particular experiment 22 and 24) show that if too many weak features are dropped at once, it might have some impact on model learning.

Finally, Table 21 shows the results of retaining the features that seem the most important so far: *ETtiTraceDlParUe_wbCqiCompensateCw0*, *ETtiTraceDlParUe_dlrrmPdcchCqiShift* and *ETtiTraceDlParUe_rrmPdschAvgUeTput*. One can observe from experiment 27 in Table 21 that for XGBoost it seems that only these two features *ETtiTraceDlParUe_wbCqiCompensateCw0* and *ETtiTraceDlParUe_dlrrmPdcchCqiShift* are enough to mount up the model performance to an AUC of 0.89 as compared to the AUC reference of 0.91. On the contrary, same cannot be said about MLP. MLP model does not learn well without the rest of the features and AUC drops significantly.

To summarize, the section described in detail the process and experiments that were undertaken to analyze the features importance. Some features came out to be more important as compared to others and help the model to train better. But experiments show that no feature has negative impact on model performance.

Table 20: Experiments with dropped features. Different combination of weak features were dropped and the collective impact on model performance was evaluated.

Index.	Dropped features	XGBoost		MLP	
		Balanced Accuracy (%)	AUC	Balanced Accuracy (%)	AUC
Reference		**70**	**0.91**	**57.42**	**0.82**
11	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_ModulationCw1	68.77	0.90	55.75	0.81
12	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_ModulationCw1 ETtiTraceDlParUe_rrmPdschAvgResAllocationUe	68.86	0.90	55.00	0.81
13	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_ModulationCw1 ETtiTraceDlParUe_dlrrmPdcchCqiShift	68.86	0.90	55.00	0.81
14	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_ModulationCw1 ETtiTraceDlParUe_rrmDeltaCqiCw0	68.90	0.89	54.63	0.80
15	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_ModulationCw1 EHarqParDl_rrmRecommendedMcsCw1	69.31	0.90	55.55	0.82
16	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_rrmPdschAvgResAllocationUe	69.91	0.89	56.49	0.81
17	ETtiTraceDlParUe_ModulationCw1 ETtiTraceDlParUe_rrmPdschAvgResAllocationUe	69.02	0.89	56.67	0.82
18	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_dlrrmPdcchCqiShift	67.59	0.88	54.86	0.81
19	ETtiTraceDlParUe_ModulationCw1 ETtiTraceDlParUe_dlrrmPdcchCqiShift	67.84	0.89	54.26	0.80
20	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_ModulationCw1 ETtiTraceDlParUe_rrmPdschAvgResAllocationUe ETtiTraceDlParUe_dlrrmPdcchCqiShift	68.08	0.89	54.29	0.78
21	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_ModulationCw1 ETtiTraceDlParUe_rrmPdschAvgResAllocationUe ETtiTraceDlParUe_wbCqiCompensateCw0	69.68	0.89	54.75	0.80
22	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_ModulationCw1 ETtiTraceDlParUe_rrmPdschAvgResAllocationUe EHarqParDl_rrmRecommendedMcsCw1	67.95	0.89	53.91	0.81
23	ETtiTraceDlParUe_wbCqiCompensateCw0 ETtiTraceDlParUe_averCirRgbCw0 EHarqParDl_rrmRecommendedMcsCw1	70.67	0.89	53.79	0.78
24	ETtiTraceDlParUe_mcsIndexCw1 ETtiTraceDlParUe_ModulationCw1 ETtiTraceDlParUe_wbCqiCompensateCw0 EHarqParDl_rrmRecommendedMcsCw1 ETtiTraceDlParUe_averCirRgbCw0	71.94	0.88	52.51	0.76

Table 21: Experiments with retained features. All features were dropped except few strongs features to evaluate how good a model can perform with only strong features.

Index.	Retained features	XGBoost		MLP	
		Balanced Accuracy (%)	AUC	Balanced Accuracy (%)	AUC
Reference		**70**	**0.91**	**57.42**	**0.82**
26	ETtiTraceDlParUe_wbCqiCompensateCw0 ETtiTraceDlParUe_rrmPdschAvgUeTput	63.86	0.79	50.00	0.72
27	ETtiTraceDlParUe_wbCqiCompensateCw0 ETtiTraceDlParUe_dlrrmPdcchCqiShift	76.88	0.89	50.00	0.725
28	ETtiTraceDlParUe_wbCqiCompensateCw0 ETtiTraceDlParUe_dlrrmPdcchCqiShift ETtiTraceDlParUe_rrmPdschAvgUeTput	69.47	0.87	51.30	0.75

4.4 Optimum Window Size

In time series prediction, it is important to know how many past values (past window size) are sufficient to use in order to make a prediction in the future. Choosing the right window size can help significantly with model learning in terms of performance and speed. Using too many past values of features can increase the feature space, risk of overfitting and training time. In this section, we explore briefly how many past values are important to use for decision trees and neural networks for our dataset. In order to find the optimum window size, we will use two methods. First by using the *feature_importances_* property of trained XGBoost model, we can categorize features of highest scores into their relevant windows. Secondly we vary the window size and evaluate the model performance.

In order to categorize the important features into their relevant windows, we can use the same methodology we employed in Section 4.3. In order to explain in detail, let us assume that we are using a window size of 15. Since we have 10 features, we end up with total of 150 feature length. Let us say that feature 'f101' (feature at 101^{th} index) gets a score of l. Now since we know feature length and window size, we can easily determine that this feature belongs to a time window at index 11, meaning the feature values belongs to a time frame that happened $(t-11)$ ms ago. We can simply add the score of this feature (f101) to score of time window at $(t-11)$. This way, we can keep track of scores of each window for the last 15 ms. Moreover, for scoring, we also employ a penalty function, Equation (23), similar to Section 4.3. Although it may seem that penalizing past values will effectively make recent past values more useful and make the whole experiment biased, but results show that it does not affect the windows' contribution scores, with or without the penalty function, except making the bar graph more conspicuous. Figures 36, 37, 38, 39, 40, and 41 show the importance of windows when trained with XGBoost with window sizes of 5, 10, 15, 20, 25, and 50 respectively. In the figures, a given index i shows past window of $t-i$ ms, while the score simply shows their relative importance to XGBoost while making predictions. One can observe that recent feature values in the near past are more important as compared to feature values that happened long time ago. Although, we do observe some peaks in the far past too; this indicates that few contributions from feature values in the far past can be potentially useful for the classifier.

In order to verify the results from the graphs further, XGBoost and MLP were trained with varying window sizes, from 5 to 50. A summary of results can be seen in Table 22. One can observe from Table 22 that smaller windows result in small decrease in AUC while bigger windows help to increase performance. For research question at hand, it is practical to use smaller windows instead of bigger ones. The reason being that in LTE and 5G, communication takes place at 1ms interval (transmission time interval) and time latency is crucial. Therefore, relying too much on past values (one also needs to take into account the time taken for preprocessing) can become impractical. From Table 22, window size of 10 and 15 seem practical options.

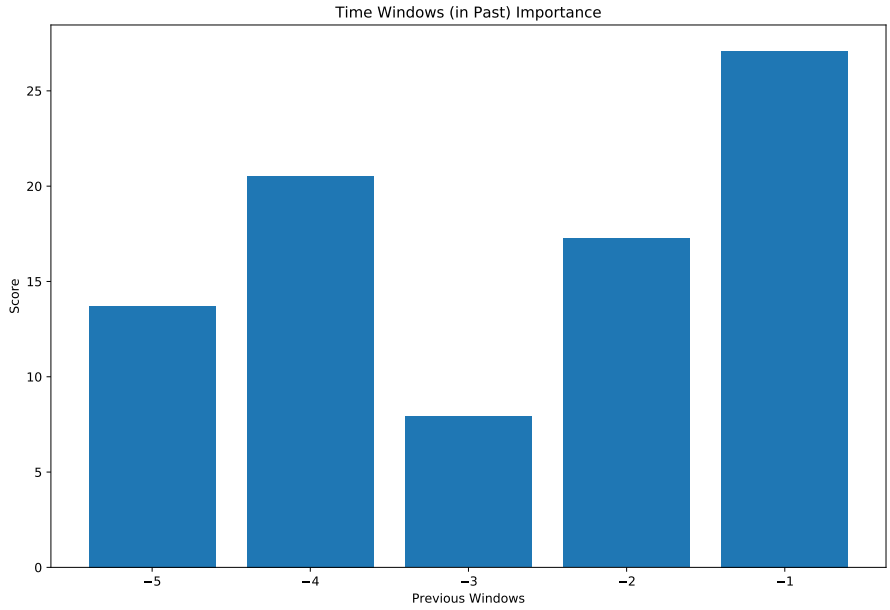


Figure 36: Windows importance with window size of 5. Using a small past window size, we observe that all 5 past values of features are important for XGBoost.

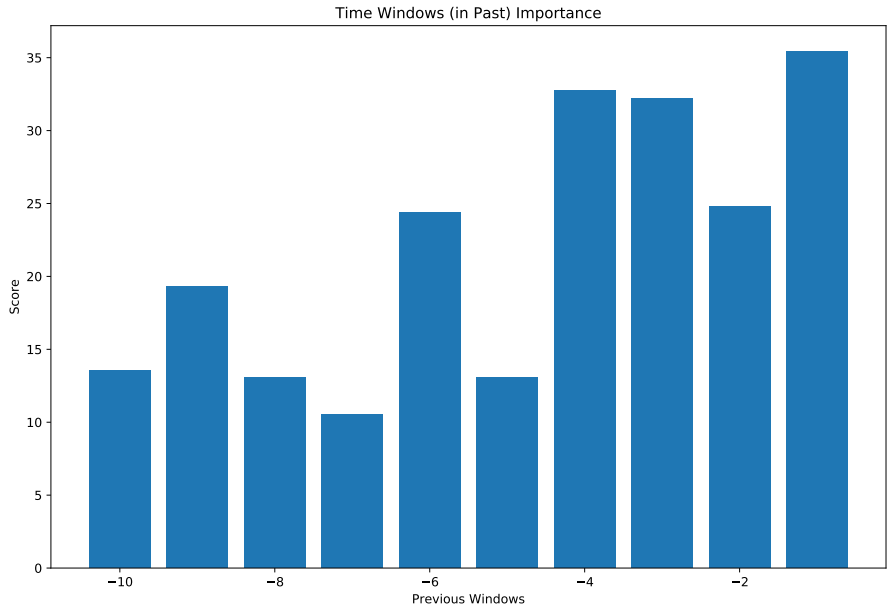


Figure 37: Windows importance with window size of 10. Last 4 values of features from the past are the most important. Feature values at $t < (t - 4)$ ms are relatively less important.

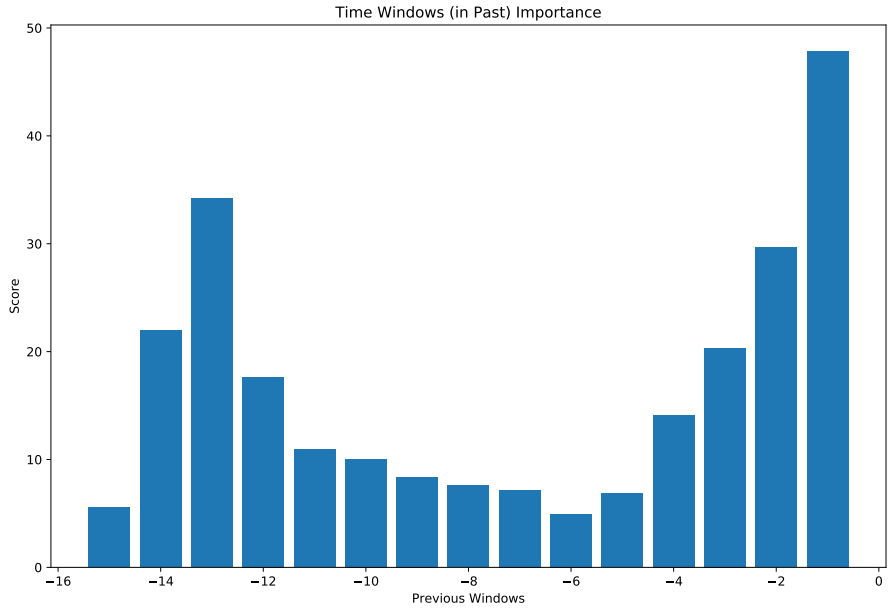


Figure 38: Windows importance with window size of 15. Not only last 4 values of features are important for XGBoost but also feature values from $t-12$ to $t-14$ ms are important.

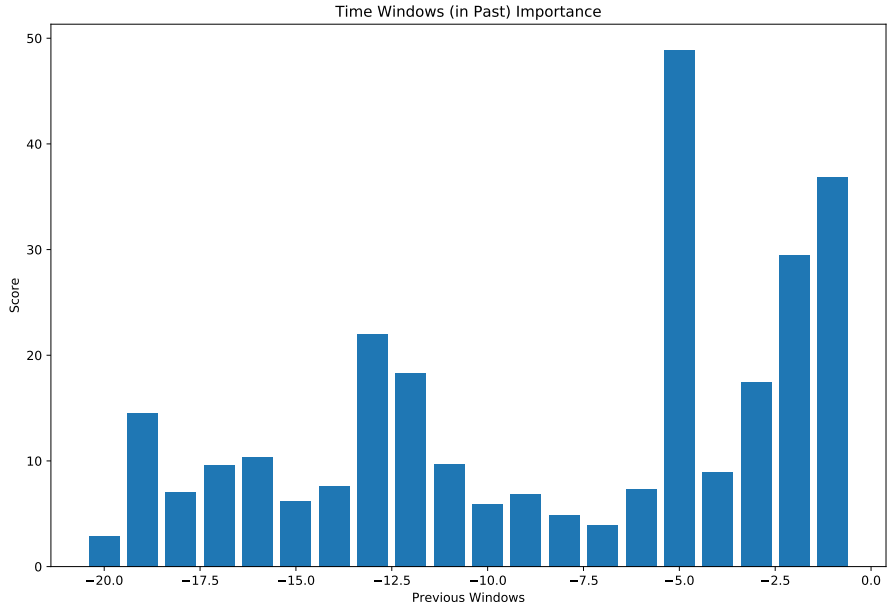


Figure 39: Windows importance with window size of 20. Recent past values of features upto $t-5$ ms are the most important.

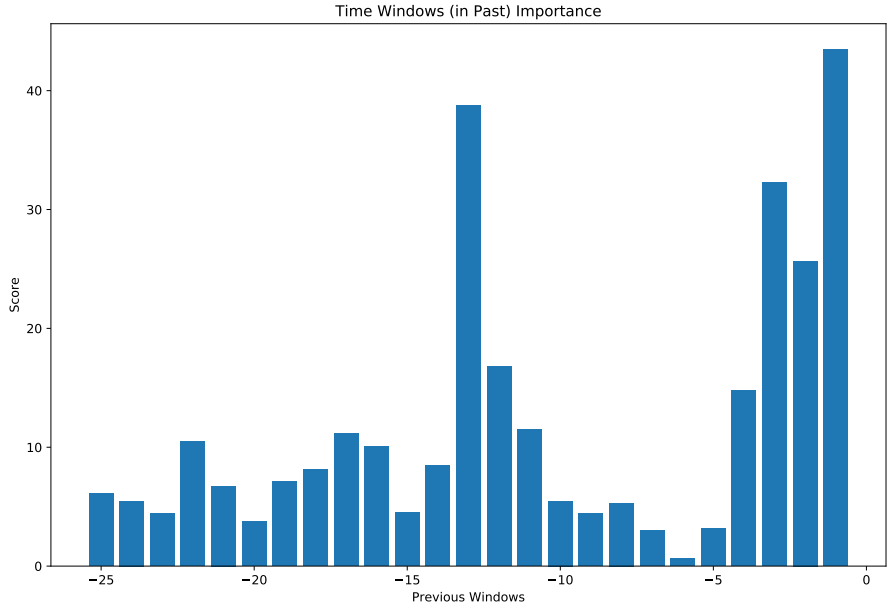


Figure 40: Windows importance with window size of 25. Not only the feature values upto $t-4$ ms are important but also some feature values that happened 12 to 14 ms ago are important. This is similar to the bar graph obtained with window size of 15.

Table 22: AUC score with different window sizes.

Model	**Window Size**					
	5	10	15	20	25	50
XGBoost	0.88	0.90	0.91	0.92	0.91	0.91
MLP	0.79	0.82	0.82	0.81	0.82	0.85

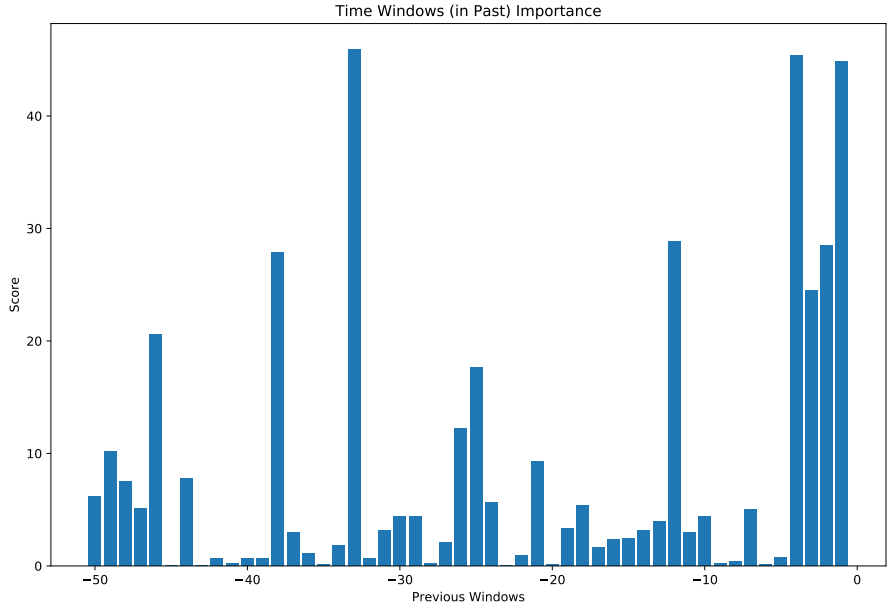


Figure 41: Windows importance with window size of 50. Again, recent 4 contribution from feature values are really important. Some peaks are observed in the far past ($t < (t - 30)$) which indicates that it might be useful to use some feature values from the far past.

5 Conclusion and Discussion

In this thesis, we explored in detail if it is possible to predict successful transmission of a given data packet in cellular networks. Our aim was to build an intelligent predictor for packet failures based on the network data as opposed to naïve approaches commonly used in wireless networks. In order to solve this problem, we used machine learning and deep learning algorithms to train models on the sequential data. Machine learning algorithm that we use for training are: long short-term memory (LSTM), gated recurrent units (GRU), multilayer perceptrons (MLP), and boosting decision trees (XGBoost). All of these algorithms are state-of-the-art and have been proven to be really useful in solving sequential tasks and time series prediction problems.

The dataset available for training was highly skewed, meaning class labels were imbalanced. Moreover, the class label of interest was highly under-represented and had significantly fewer samples. In order to deal with this issue, while training neural networks and boosting decision trees, class weights were used. This was to make sure that model penalizes false detection of under-represented class labels more than the dominant class. Our results in chapter 4 show that class weights parameter was quite useful, both in neural networks and XGBoost, in improving model performance.

In deep learning, tuning neural networks can be really difficult problem since it involves a lot of hyperparameters configuration. In order to get optimal values of hyperparameters, a systematic approach was adopted, where each parameter was changed and its impact on the model performance was evaluated. Among different activation functions used in MLP, tanh and ELU (exponential linear units) provided the best results. In LSTM and GRU recurrent networks, tanh activation function is used by default as per their architectures, so it was not changed. Regarding optimal optimizer, *Adam* and *Adamax* performed the best. It was no surprise since many of the deep learning research problems use *Adam* these days. *RMSProp* also provided decent results while *SGD* performed the worst and model did not learn with *SGD* optimizer.

Learning rate of the optimizing algorithm is usually one of the most experimented with hyperparameter while training neural networks. In deep learning, one needs to decrease learning rate of optimizing algorithm to make sure that model converges properly. If larger learning rate is used, model might have difficulty in convergence since optimizer will start taking bigger steps and might never find a minimum. One of the fundamental issues that comes with decreasing learning rate is significant increase in training time. In our results, we do not change learning rate much, rather we tune batch size as mentioned in this research paper by Smith et al. (2017). The idea is instead of decreasing the learning rate, one could increase the batch size and get similar model performance with far less training time. While adhering to this principle, different batch sizes were experimented with. Using single datafile, 96 and 256 were found out to be good batch sizes for LSTM/GRU and MLP; whereas with combined dataset, 8192 and 20000 were the optimal batch size for LSTM/GRU and MLP (according to our results).

Model overfitting is one of the main causes of poor model performance and must be accounted for while training machine learning algorithms. There can be several

causes of model overfitting; using increased feature space and complex models for training are one of the known causes, described below briefly:

1. In order to avoid overfitting by features, several features were dropped to evaluate the impact of each feature on the model performance. Section 4.3 described in detail the process and experiments that were undertaken to analyze the features importance. Some features came out to be more important as compared to others and helped the model to learn better. But it is difficult to draw a line between weak and strong features; some features might be more helpful for decision trees while others for neural networks. For our particular problem, since no feature had a negative impact on model performance and the focus is more towards a better model performance instead of reducing training time, it was decided to retain all the features.

2. While training a machine learning model, it is always preferable to get good model accuracy using a simpler model since a complex model will learn the training data too well and will not be able to generalize its learning to the unseen dataset. Therefore, it is always preferable to use a simple model; and if simple model does not work then one needs to find a balance between a slightly complex model that barely gets the job done and a complicated model that might lead to overfitting. While tuning number of neurons/units/trees, layers or depth in MLP, LSTM/GRU and XGBoost, this principle was strictly adhered to. Moreover, early stopping and dropouts were used in neural networks to make sure that model does not overfit. In XGBoost, *min_child_weight* and *gamma* parameters were used for regularization. Results in chapter 4 show in detail that these methods were extremely useful to ensure that model learning is not biased towards training data and is able to generalize on test data.

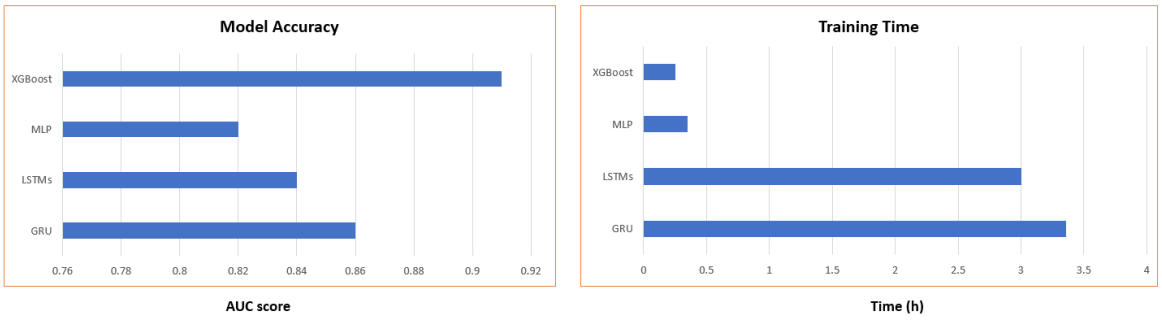


Figure 42: Comparison of models' accuracy and training time on single datafile. XGBoost clearly outperforms neural networks both in AUC score and training time.

Figure 42 and 43 shows the comparison of best results obtained from each model on single dataset and combined dataset respectively. One can note that using smaller dataset (single datafile), XGBoost outperforms neural networks both in accuracy and training time. But as the dataset size becomes ten times larger, XGBoost accuracy drops and takes longer time to train. Contrary to this, if we analyze LSTM results,

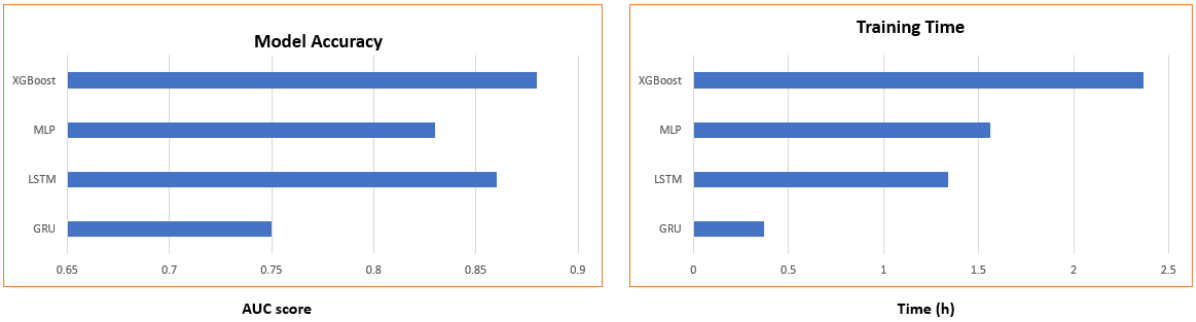


Figure 43: Comparison of models' accuracy and training time on combined dataset. XGBoost model performance provides the highest AUC score although it takes significantly longer time to train on combined dataset.

we see that LSTMs accuracy increases and training time decreases as the dataset gets 10x larger. This is because neural networks train better when the training samples increase (Sun et al., 2017). Decrease in training time of LSTM on bigger dataset is associated with increased batch size. Using a bigger batch size on smaller dataset did not result in good accuracy score, therefore we had to use smaller batch size of 96 and it resulted in 3 to 7 hours of training time. Compared to this, on bigger dataset, using a bigger batch size did not cause any problems and we opted for a batch size of 20K, even 50K batch size provided good results. On the other hand, Multilayer perceptron model performance increased slightly on bigger dataset and training time increased too. Lastly, GRUs performance significantly decreased when it was trained on bigger dataset, which is quite odd. Different hyperparameter configurations were experimented, but GRU performance remained poor for bigger dataset.

From Figure 42 and 43, one can note that given the dataset at hand, XGBoost is the best performing ML algorithm, on both single and combined dataset. The results indicate that data packet failures in wireless networks are not random and can be predicted with sufficient accuracy. For example, using XGBoost, 42% of packet failures can be predicted at the expense of 0.67% false alarms. Compared to naïve approaches used in cellular communication to avoid packet failures (like packet duplication), our solution based on intelligent packet error prediction indicates promising practical applications in cellular network for enhanced radio network performance.

Although our research shows very promising results for predicting packet failures, there is plenty of room for future work in this research problem. In this thesis, using previous values of parameters, we only determine the packet failures in the next transmission time interval (TTI). It would be useful to conduct further experiments to analyze if packets in the second, third (and so on) TTIs can be predicted or not. Moreover, in this thesis, only limited machine learning algorithms were experimented with on the given dataset. Potential state-of-the-art machine learning algorithms that can be used for this task include Transformer (Vaswani et al., 2017), 2D convolutional based neural network with causal convolution (Elbayad et al., 2018), and temporal convolutional networks (TCN) (Bai et al., 2018). Attention based sequence models and simple convolutional architecture for sequence modeling have been reported to perform better than traditional recurrent neural networks on machine translation (Vaswani et al., 2017; Bai et al., 2018). One could try these relatively more advanced algorithms and benchmark their performance against RNNs and XGBoost.

References

Ali-Yahiya, T. (2011). *Understanding LTE and its Performance*. Springer Science & Business Media.

Alpaydin, E. (2009). *Introduction to Machine Learning*. MIT press.

Anthimopoulos, M., Christodoulidis, S., Ebner, L., Christe, A., and Mougiakakou, S. (2016). Lung pattern classification for interstitial lung diseases using a deep convolutional neural network. *IEEE Transactions on Medical Imaging*, 35(5):1207–1216.

Bai, S., Kolter, J. Z., and Koltun, V. (2018). An empirical evaluation of generic convolutional and recurrent networks for sequence modeling. *arXiv preprint arXiv:1803.01271*.

Bauer, E. and Kohavi, R. (1999). An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine Learning*, 36(1-2):105–139.

Bengio, Y., Simard, P., and Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166.

Brodersen, K. H., Ong, C. S., Stephan, K. E., and Buhmann, J. M. (2010). The balanced accuracy and its posterior distribution. In *2010 20th International Conference on Pattern Recognition*, pages 3121–3124. IEEE.

Burton, H. O. and Sullivan, D. D. (1972). Errors and error control. *Proceedings of the IEEE*, 60(11):1293–1301.

Chen, T. and Guestrin, C. (2016). XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIG International Conference on Knowledge Discovery and Data Mining*, pages 785–794.

Chen, X.-W. and Liu, M. (2005). Prediction of protein–protein interactions using random decision forest framework. *Bioinformatics*, 21(24):4394–4400.

Cho, K., Merrienboer, B. V., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., and Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.

Chuang, W. T. and Yang, J. (2000). Extracting sentence segments for text summarization: A machine learning approach. In *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, pages 152–159. ACM.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv e-prints arXiv:1412.3555*.

Clevert, D.-A., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (ELUs). *arXiv e-prints*.

Davida, G. I. and Reddy, S. M. (1972). Forward-error correction with decision feedback. *Information and Control*, 21:117–133.

Deng, L., Hinton, G., and Kingsbury, B. (2013). New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8599–8603. IEEE.

der Maaten, L. V. and Hinton, G. (2008). Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605.

Devlin, J., Zbib, R., Huang, Z., Lamar, T., Schwartz, R., and Makhoul, J. (2014). Fast and robust neural network joint models for statistical machine translation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1370–1380.

El-Sayed, M. and Jaffe, J. (2002). A view of telecommunications network evolution. *IEEE Communications Magazine*, 40(12):74–81.

Elbayad, M., Besacier, L., and Verbeek, J. (2018). Pervasive attention: 2D convolutional neural networks for sequence-to-sequence prediction. *arXiv preprint arXiv:1808.03867*.

Enns, F. and O'Hare, P. J. (1991). Packet framing using cyclic redundancy checking.

Fawcett, T. (2006). An introduction to ROC analysis. *Pattern Recognition Letters*, 27(8):861–874.

Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics & Data Analysis*, 38(4):367–378.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 315–323.

Goldberg, Y. (2016). A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57:345–420.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

Guyon, I. and Elisseeff, A. (2003). An introduction to variable and feature selection. *Journal of Machine Learning Research*, 3(Mar):1157–1182.

Haselsteiner, E. and Pfurtscheller, G. (2000). Using time-dependent neural networks for EEG classification. *IEEE Transactions on Rehabilitation Engineering*, 8(4):457–463.

Ho, T. K. (1995). Random decision forests. In *Proceedings of 3rd International Conference on Document Analysis and Recognition*, volume 1, pages 278–282. IEEE.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

Hong, W.-C. (2008). Rainfall forecasting by technological machine learning models. *Applied Mathematics and Computation*, 200(1):41–57.

Hyndman, R. J. and Athanasopoulos, G. (2019). *Forecasting: Principles and Practice*. OTexts: Melbourne, Australia, 2nd edition.

Johansson, N. A., Y.-P. Eric Wang, E., Eriksson, E., and Hessler, M. (2015). Radio access for ultra-reliable and low-latency 5G communications. In *2015 IEEE International Conference on Communication Workshop (ICCW)*, pages 1184–1189. IEEE.

Jolliffe, I. (2011). *Principal Component Analysis*. Springer.

Karunasinghe, D. S. and Liong, S.-Y. (2006). Chaotic time series prediction with a global model: Artificial neural network. *Journal of Hydrology*, 323(1-4):92–105.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kleinbaum, D. G., Dietz, K., Gail, M., Klein, M., and Klein, M. (2002). *Logistic Regression*. Springer.

Kohavi, R. and Sommerfield, D. (1995). Feature subset selection using the wrapper method: Overfitting and dynamic search space topology. In *Knowledge Discovery in Databases (KDD)*, pages 192–197.

Kotsiantis, S. B., Zaharakis, I., and Pintelas, P. (2007). Supervised machine learning: A review of classification techniques. *Emerging Artificial Intelligence Applications in Computer Engineering*, 160:3–24.

Kriesel, D. (2007). *A Brief Introduction to Neural Networks*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, pages 1097–1105.

Ku, G. (2011). Resource allocation in LTE. *Adaptive Signal Processing and Information Theory Research Group*.

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86.

Lai, S., Xu, L., Liu, K., and Zhao, J. (2015). Recurrent convolutional neural networks for text classification. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Langlois, D., Chartier, S., and Gosselin, D. (2010). An introduction to independent component analysis: InfoMax and FastICA algorithms. *Tutorials in Quantitative Methods for Psychology*, 6(1):31–38.

Lisboa, P. J. and Taktak, A. F. (2006). The use of artificial neural networks in decision support in cancer: A systematic review. *Neural networks*, 19(4):408–415.

Love, R. T., Bachu, R. S., Classon, B. K., Nory, R., Stewart, K. A., and Sun, Y. (2008). Channel quality indicator for time, frequency and spatial channel in terrestrial radio access network.

Pascanu, R., Mikolov, T., and Bengio, Y. (2012). On the difficulty of training recurrent neural networks. *arXiv e-prints*.

Powers, D. M. (2011). Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. *Journal of Machine Learning Technologies*, pages 37–63.

Prechelt, L. (1998). Early stopping-but when? In *Neural Networks: Tricks of the Trade*, pages 55–69. Springer.

Proakis, J. G. (2001). *Digital Communications*.

Rao, J. and Vrzic, S. (2018). Packet duplication for URLLC in 5G: Architectural enhancements and performance analysis. *IEEE Network*, 32(2):32–40.

Rish, I. et al. (2001). An empirical study of the naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46.

Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6).

Sheppard, C. (2017). *Tree-based Machine Learning Algorithms: Decision Trees, Random Forests, and Boosting*.

Smith, S. L., Kindermans, P.-J., Ying, C., and Le, Q. V. (2017). Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*.

Sun, C., Shrivastava, A., Singh, S., and Gupta, A. (2017). Revisiting unreasonable effectiveness of data in deep learning era. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 843–852.

Sun, Z.-L., Choi, T.-M., Au, K.-F., and Yu, Y. (2008). Sales forecasting using extreme learning machine with applications in fashion retailing. *Decision Support Systems*, 46(1):411–419.

Sutskever, I., Vinyals, O., and Le, Q. V. (2014). Sequence to sequence learning with neural networks. In *Advances in Neural Information Processing Systems 27*, pages 3104–3112.

Tang, Z. and Fishwick, P. A. (1993). Feedforward neural nets as models for time series forecasting. *ORSA Journal on Computing*, 5(4):374–385.

Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1):267–288.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Vens, C., Struyf, J., Schietgat, L., Džeroski, S., and Blockeel, H. (2008). Decision trees for hierarchical multi-label classification. *Machine Learning*, 73(2):185.